MASTER EN SCIENCES INFORMATIQUES

Traduction de TVL en SMT-LIBv2

Vanderveken, Thibault

Award date:
2011

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix, Namur

Faculté d'informatique

Année Académique 2010 - 2011

Traduction de TVL en SMT-LIBv2

Thibault Vanderveken

Mémoire présenté en vue de l'obtention du grade de Master en Sciences Informatiques

Préface

Les *feature models* jouent un rôle clé dans la capture des similitudes et de la variablité des lignes de produits. Depuis la notation originale FODA, les *feature models* ont connu plusieurs extensions et notations dont *TVL*, qui combine la plupart des extensions existantes, ce qui lui confère une grande expressivité. Depuis des années, les automatisations sur les *feature models* comme la vérification de la satisfiabilité font l'objet de recherche et l'on a déjà proposé de nombreuses solutions. Dans ce document, on fournit une solution pour vérifier la consistance de feature model en *TVL*. Pour ce faire, on a utilisé *SMT-LIBv2* qui définit un langage commun d'entrée et de sortie pour les solveurs SMT. Il est également question des différentes possiblités de traduction ainsi que des limites de l'approche.

Feature models play a crucial role within *Software Product Lines* : they allow the capture of commonalities and variability among the various constituent products. Different notations and extensions are used to represent *feature models* among which *TVL*, which combines most of the existing extensions, making it a highly expressive textual language. In recent years, *feature model* automatisations such as satisfiability checks have been the subject of research ; many solutions have been proposed. In this document, a solution is put forward in order to check *feature model* consistency in *TVL*. To this end, *SMT-LIBv2* was used because it defines a common input and output language for SMT solvers. The various possibilities regarding translation and the limitations of the approach are also discussed.

Remerciements

Je tiens à remercier mon promoteur, Patrick Heymans, pour m'avoir donné l'opportunité de partir quatre mois à l'étranger ainsi que pour son aide précieuse tout au long du stage et du mémoire. Ce stage m'a permis, à la fois, de parfaire mon anglais et de découvrir un domaine informatique qui m'était jusqu'alors méconnu. Je tiens à remercier Krzysztof Czarnecki de m'avoir accueilli durant mon stage dans le *Generative Software Development Lab* [a] ainsi que toute son équipe, et en particulier : Michal Antkiewicz, Kacper Bak, Leonardo Passos et Marko Novakovic.

Je tiens également à remercier Arnaud Hubaux pour ses conseils durant mon stage ainsi que Raphaël Michel pour son aide et ses recommandations lors de l'élaboration et la rédaction du mémoire.

Finalement, j'aimerais remercier ma famille et mes amis qui m'ont toujours soutenu tout au long de mes études ainsi que lors des différents projets que j'ai entrepris. Je tiens à remercier en particulier ces personnes pour leur aide, leurs observations et leur bonne humeur : Guillaume Benats, Julie Craps, Thomas Eugène, Nicole Gregoire, Sarah Pakdaman, Johan Schuiten, Thibault Smets et Alexandre Vettas.

a. http ://gsd.uwaterloo.ca/

Table des matières

Table des figures

Liste des tableaux

Table des codes sources

Acronymes et abréviations

BDD Binary Decision Diagram.

CSP Constraint Satisfaction Problem.

DAG Directed Acyclic Graph.

DL Description Logic.

FD Feature Diagram.

FM Feature Model.

LALR Look-Ahead Left Recursive.

NPC NP-Complete.

OWL Web Ontology Language.

SAT Satisfiability.

SMT Satisfiability Modulo Theories.

SMT-LIB Satisfiability Modulo Theories Library.

SPL Software Product Line.

TVL Textual Variability Language.

UML Unified Modeling Language.

XML Extensible Markup Language.

1 Introduction

De nos jours, les *feature models* font partie intégrante du développement de *Software Product Line* (*SPL*). En effet, ils permettent de capturer les similitudes et la variabilité qui apparaissent dans une *SPL*. Les *feature models* sont des modèles d'information où un ensemble de produits est représenté par un ensemble de features dans un modèle unique [10]. Ils sont fréquemment utilisés tout au long du processus de développement du produit et l'on peut les utiliser en tant qu'entrée de données dans le but de produire divers documents. [41]

La modélisation de features fut proposée pour la première fois par Kang *et al.* dans la méthode *FODA* (*Feature-Oriented Domain Analysis*) [24]. Depuis, un grand nombre de notations sont apparues. Celles-ci sont, pour la plupart, visuelles (graphiques), mais celle qui fait l'objet de cette étude est textuelle : *TVL* (*Textual Variability Language*) [12, 13]. Ce langage tente d'améliorer la conception de *feature models* à grande échelle pour lesquels la représentation graphique peut poser problème. Il fournit également aux développeurs un langage textuel compréhensible permettant de rendre la modélisation plus facile [13]. Il permet également l'emploi d'attributs, de cardinalités ainsi que l'utilisation de contraintes complexes. Tout ceci fait de lui une alternative intéressante aux notations graphiques.

L'analyse automatique de *feature models* consiste à extraire des informations d'un *feature model* en utilisant des mécanismes automatiques [10]. Des automatisations telles que la vérification de la satisfiabilité du modèle, la recherche du nombre de produits réalisables et la configuration interactive/non-

interactive de produits sont désormais tous possibles. On remarque que la variabilité dans les *SPLs* augmente fortement et cela se répercute évidemment dans le nombre de produits potentiels [8]. Par conséquent, on a besoin de méthodes et d'outils adaptés permettant ces automatisations. Bien que l'on ait déjà proposé beaucoup d'analyses (vue d'ensemble par Benavides *et al.* [10, 8]), ce domaine de recherche reste fort actif et prend de l'importance aussi bien auprès des utilisateurs que des chercheurs de la communauté *SPL* [10].

Actuellement, beaucoup d'analyses prennent en compte les *feature models* classiques et ceux basés sur les cardinalités. Néanmoins, les analyses concernant les attributs sur un *feature model* sont limitées [10]. C'est pourquoi l'on a décidé de concevoir une traduction du langage *TVL* vers un langage *SMT*. Pour ce faire, on a utilisé le langage *SMT-LIBv2* [4]. Il s'agit d'un langage standardisé qui sert d'entrée aux solveurs *SMT* [44] et qui facilite la recherche et le développement dans le domaine *SMT*.

1.1 Questions de recherches

Ce document permettra dans son ensemble de répondre à la question suivante :

> Comment peut-on traduire un *feature model* en un problème *SMT* (dans notre cas, le langage *TVL* vers le langage *SMT-LIBv2*) ? Et quelles sont les caractéristiques de cette traduction ?

Pour y arriver, la recherche des réponses aux questions suivantes m'a semblé pertinente :

Q1 Comment peut-on définir un *feature model* ? Quelle a été son évolution au fil du temps ? Quelles sont les différentes opérations que l'on peut lui appliquer ?

Q2 Comment peut-on définir la satisfiabilité ? Comment peut-on la vérifier ?

Q3 En quoi consistent les problèmes *SMT* (*Satisfiability Modulo Theories*) ?

Q4 Quelles sont les caractéristiques du langage *TVL* ?

Q5 A quoi correspond la librairie *SMT-LIB* (*The Satisfiability Modulo Theories Library*) ? Quelles sont les caractéristiques du langage qu'elle propose ?

1.2 Structure du document

Dans la section 2, on aborde les différentes notions nécessaires à la compréhension de cette traduction telles que la satisfiabilité, les feature models ainsi que les *Satisfiability Modulo Theories*. Par la suite, on traite des deux langages utilisés durant cette étude : le langage d'entrée *TVL* et le langage de sortie *SMT-LIBv2*. Quant à la section 4, on y présente l'étude de cas sur laquelle je me suis basé pour la traduction.
Ensuite, dans la section 5, il est question de la traduction même des différentes structures présentes en *TVL* vers le langage *SMT-LIBv2*. La section 6 est consacrée à l'implémentation de cette traduction ainsi que les différents processus qui entrent en jeu pour l'accomplir.
Dans la section 7, la conclusion fait le point sur le travail effectué et propose des améliorations futures possibles.

2 Etat de l'art

Dans cette section, on fait le point sur la littérature relative aux objectifs de ce mémoire. On détaille donc la notion de feature model, son évolution, les outils existants et les automatisations possibles. Il sera également question de la satisfiabilité ainsi que des Satisfiability Modulo Theories.

2.1 Feature, *Feature Diagram* et *Feature Model*

Cette partie a été inspirée par divers documents [9, 10, 8, 41, 17, 33].

Un *feature model* (FM) représente les différentes configurations possibles d'une ligne de produits (*Software Product Line – SPL*) en termes de features et de relations entre celles-ci. Ce terme "*feature model*" fut introduit pour la première fois par Kang *et al.* [24] dans la méthode FODA (*Feature-Oriented Domain Analysis*). Un *feature model* peut être utilisé à différents niveaux de développement tels que l'ingénierie des exigences, la définition d'architecture ou la génération de code (programmation orientée feature) [9].

Selon Kang *et al.* [24], « une feature est définie comme un aspect, une qualité ou une caractéristique visibles par l'utilisateur et importants ou spéciaux dans un système logiciel ou système ». En 2000, Czarnecki et Eisenecker [15] proposent une autre définition : « une feature est une propriété d'un système qui est importante pour une partie prenante et qui est utilisée pour capturer les similitudes ou les différences des systèmes d'une famille ». Cette deuxième définition permet de prendre en compte non seulement les exigences fonctionnelles, mais aussi les non-fonctionnelles [17].

Un *feature model* est généralement représenté de manière graphique. Il est composé d'une structure en arbre où les nœuds représentent des features et où les arcs représentent une décomposition hiérarchique de features. Il est donc composé de :

- relations entre une feature parent et des features enfants (sous-features) ;
- contraintes *cross-tree* (aussi appelées *left-over*) qui sont typiquement des implications ou exclusions de la forme : "Si F est incluse, la feature X doit également être incluse (ou exclue)".

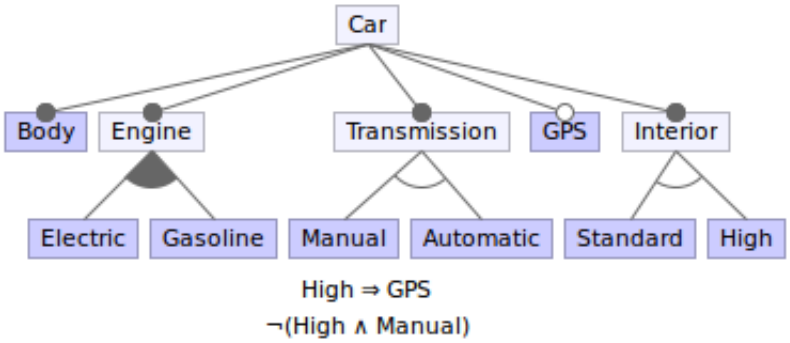
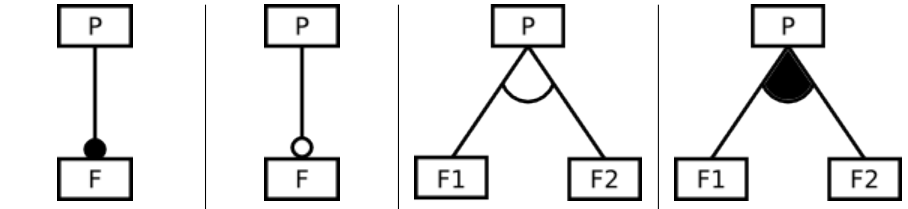


FIGURE 1 – Exemple de *feature model*

Le *feature model* est donc composé d'un diagramme, appelé *feature diagram*, auquel on peut ajouter des contraintes *cross-tree* de manière graphique (fig. 3) ou textuelle. Le *feature diagram* offre une notation simple et intuitive pour représenter les points de variation indépendemment des mécanismes d'implémentation tels que l'héritage ou l'agrégation [16]. Notons également que la feature racine (*root*) représente le concept (*Car*) qui sera défini. Il sera présent dans tous les produits composant la *SPL*.

Depuis la proposition originale FODA, on a proposé un grand nombre

d'extensions et de variantes ont été proposées pour les *feature models* [16]. On peut trouver davantage de détails dans l'article de Schobbens *et al.* [34]. Parmi celles-ci, la notation proposée par Czarnecki et Eisenecker [15] est la plus compréhensible et flexible. De plus, c'est la plus citée [8]. Par conséquent, nous allons la détailler ci-dessous.



(a) Mandatory (b) Optional (c) Alternative (d) Or

FIGURE 2 – Notation de *feature model* (Czarnecki [15])

Sur la figure 1 (partiellement inspirée de [9]), on voit un exemple de *feature model* qui représente une voiture (*Car*). Ce genre de *feature model* peut être utilisé dans l'industrie automobile pour spécifier et construire des logiciels pour une voiture configurable [9]. En observant le modèle au premier niveau, on remarque que chaque voiture (**Car**) est constituée d'un châssis (**Body**), d'un moteur (**Engine**), d'un intérieur (**Interior**) et se voit optionnellement équipée d'un GPS (*Global Positioning System*) (**GPS**). Plus on descend dans l'arbre, plus la description du produit devient spécifique.

On note quatre types de relation dans un *feature model* classique :

Mandatory. Une sous-feature a une relation obligatoire (*Mandatory*) avec son parent lorsque l'enfant est inclus dans le produit dès que sa feature parente apparaît. Dans notre exemple (fig. 1), la feature **Body** est obligatoire dans une voiture.

Optional. Une sous-feature est dite optionelle (*Optional*) lorsqu'elle peut apparaître ou non si sa feature parente apparaît. Dans notre exemple (fig. 1), la feature **GPS** est optionnelle dans une voiture.

Alternative. Une sous-feature dans une relation alternative (*Alternative*) peut être présente dans un produit si sa feature parente est incluse. Dans ce cas, seulement une feature sur l'ensemble des enfants doit être présente. Dans notre exemple (fig. 1), la feature transmission (**Transmission**) ne peut être que manuelle (**Manual**) ou automatique (**Automatic**).

Or. Une sous-feature dans une relation ou (*Or*) peut être présente dans un produit si sa feature parente est incluse. Dans ce cas, au moins une feature sur l'ensemble des enfants doit être présente. Dans notre exemple (fig. 1), le feature moteur (**Engine**) peut être soit électrique (**Electric**), soit à essence (**Gasoline**), soit les deux (ce qui correspond à un moteur hybride).

Comme mentionné précédemment, en plus des relations parentales entre features, des contraintes *cross-tree* entre les features peuvent exister :

Requires. Si une feature X requiert une feature Y, l'inclusion de X dans le produit requiert l'inclusion de Y dans le produit. Evidemment, l'inclusion de Y ne requiert pas forcément la présence de X. Dans notre exemple (fig. 1), la feature **High** requiert la feature **GPS**. Cela signifie que lorsque **High** est présent dans le produit, **GPS** doit forcément s'y retrouver.

Excludes. Si une feature X exclut une feature Y, les deux features ne peuvent être présentes dans le même produit. Dans notre exemple

(fig. 1), si nous choisissons la feature **Manual**, nous ne pouvons pas choisir la feature **High** et inversément.

Griss *et al.* [23] ont également introduit une notation graphique pour les contraintes *cross-tree*. Celle-ci est présentée sur la figure 3.

(a) Requires (b) Excludes

FIGURE 3 – Notation de contraintes *cross-tree*

Par la suite, on a proposé des contraintes plus complexes sous la forme de formules propositionnelles [6]. Si A, B, C, D sont des features, nous pouvons, par exemple, avoir $(A \wedge B \wedge D) \rightarrow not\ C$.
Czarnecki *et al.* [27] définissent le ratio des contraintes *cross-tree* (*cross-tree constraint ratio* (CTCR)) comme la comparaison entre le nombre de features apparaissant dans les contraintes et le nombre de features contenues dans le diagramme. Dans notre exemple illustré sur la figure 1, le ratio est de 25% (3/12). C'est un des facteurs permettant de déterminer la complexité d'un *feature model*.

Un produit individuel conforme à un *feature model* est spécifié tel un ensemble de features [27]. Par exemple, l'ensemble $S_1 = \{$`Car, Body, Engine, Gasoline, Transmission, Manual, Interior, Standard`$\}$ représente un véhicule manuel à essence ayant un intérieur standard. Par contre, l'ensemble $S_2 = \{$`Car, Body, Engine, Transmission, Automatic, Interior, High`$\}$, n'est pas valide. En effet, on observe deux éléments non conformes au modèle. En premier lieu, la présence de **Engine** nécessite la présence de soit **Electric**, soit **Gasoline**. Deuxièmement, la présence du feature **High** nécessite

la présence du **GPS** (contrainte *cross-tree*).

2.1.1 *Feature model* basé sur les cardinalités

D'autres *feature models* [16, 18, 32] sont basés sur les cardinalités (comme les multiplicités en *UML*). La motivation principale est constituée des applications pratiques ainsi que de l'enrichissement des concepts existants. Voici les nouvelles relations introduites :

Feature cardinality. Les features obligatoires et optionnelles sont généralisées : le nombre de fois qu'une feature apparaît est déterminé par sa cardinalité. Par conséquent, la relation *mandatory* est équivalente à la cardinalité $< 1..1 >$ et la relation *optional* est équivalente à la cardinalité $< 0..1 >$.

Group cardinality. Un *group cardinality* est un intervalle dénoté par $< n..m >$, où n dénote la borne inférieure et m la borne supérieure, limitant le nombre d'enfants qui peuvent faire partie du produit lorsque le parent est sélectionné. Par conséquent, la relation *alternative* est équivalente à la cardinalité $< 1..1 >$ et la relation *or* est équivalente à la cardinalité $< 1..N >$ où N dénote le nombre de features dans la relation.

Un exemple de *feature model* basé sur la cardinalité ainsi qu'une légende sont illustrés sur la figure 4 [17].

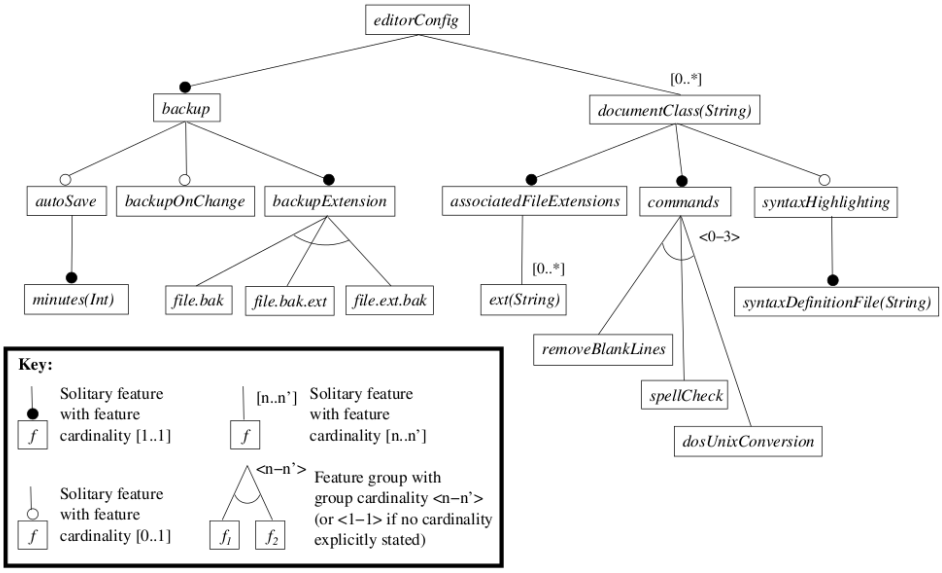


FIGURE 4 – Notation pour *feature model* basé sur les cardinalités (Czarnecki [16])

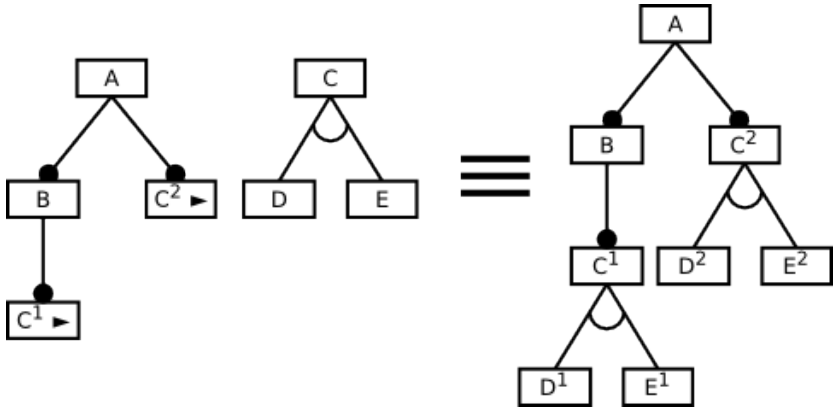


FIGURE 5 – Exemple de réference dans un *feature model*

Notons que cette notation n'autorise pas l'emploi de *feature cardinality* sur des features qui font partie d'un groupe de features étant donné que cette relation a lieu entre un parent et une feature seule. Czarnecki *et al.* [16, 19] proposent également la notion de référence qui permet de réutiliser

ou de modulariser des *features models*. Un exemple est présenté sur la figure 5.

Un tableau comparatif des différentes notations graphiques (FODA, Czarnecki Notation [15], Czarnecki Notation [16]) est disponible dans l'article de Czarnecki [16].

2.1.2 *Feature model* étendu

Un *feature model* permet de représenter la variabilité dans une *SPL*. On peut néanmoins trouver de l'intérêt à étendre les *feature models* en ajoutant de l'information sur les features. Cette information est représentée par ce que l'on appelle des attributs de feature (*feature attributes*). Ce genre de modèle, dans lequel de l'information supplémentaire est incluse, s'appelle *feature model* étendu [8, 16, 18, 19].
En réalité, la méthode FODA [24] avait déjà envisagé l'intégration d'informations supplémentaires dans un feature model [9]. Par exemple, la relation entre une feature et ses attributs avait été introduite.
Il n'y a pas de norme spécifique pour définir un attribut, mais la plupart des notations dotent un attribut d'un nom, d'un domaine et d'une valeur (par exemple, un attribut mémoire qui a une valeur comprise dans le domaine des entiers où cette valeur peut représenter une quantité en mégaoctet). Sur la figure 6, on voit un *feature model* étendu qui emploie la notation de Benavides *et al.* [8].

Dans l'exemple présenté sur la figure 6, les différentes features représentent l'aspect fonctionnel de la *SPL*. Par conséquent, l'utilisation d'attributs per-

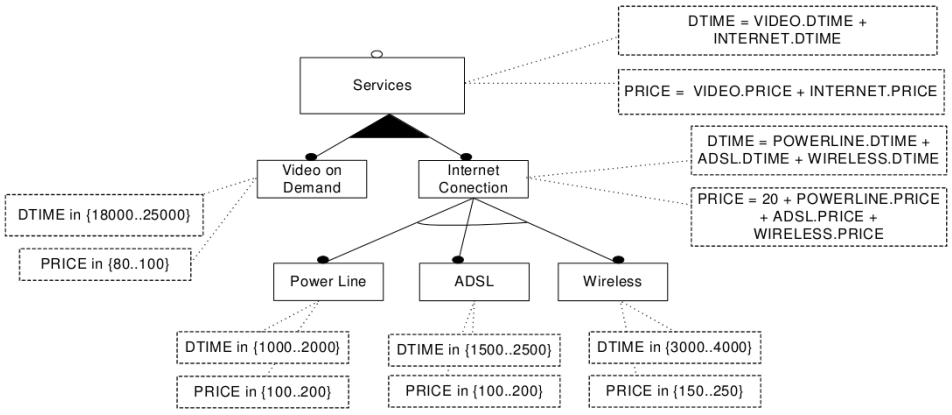


FIGURE 6 – Exemple de *feature model* étendu

met de leur ajouter des contraintes non-fonctionnelles. On observe qu'un attribut peut être représenté par un intervalle de valeur. Le prix de la vidéo à la demande, par exemple, varie entre 80 et 100. Un autre point intéressant est que les attributs peuvent être reliés entre eux (contrairement à [16]). Cela permet de rendre le prix du service (`Services`) équivalent à la somme du prix de la vidéo à la demande (`Video on Demand`)et de la connexion internet (`Internet Connection`). La connexion internet, à son tour, correspond également à la somme des attributs prix (`PRICE`) de ses enfants.

2.1.3 Automatisation

Dans cette section, nous allons décrire les opérations les plus courantes s'effectuant sur un *feature model* [9, 10, 8]. Cette liste n'est pas exhaustive.

Validité du modèle. Ceci est l'opération basique que l'on effectue sur un *feature model*. Cette opération peut également s'appeler vérification de la satisfiabilité (consistance) du modèle. Un *feature model* est satisfiable s'il existe au moins un produit qui peut être formé. On peut également noter qu'un *feature model* ne peut pas être insatisfiable s'il

n'existe pas de contraintes sur celui-ci. Cette opération est très utile lorsqu'on travaille avec de modèles de taille importante car la détection manuelle d'erreurs est faillible et prend beaucoup de temps.

Validation d'un produit. Cette opération prend comme entrée un *feature model* et un produit (ensemble de features) et renvoie une valeur qui permet de savoir si le produit fait partie des produits pouvant être formés avec le feature model.

Générer tous les produits. Cette opération consiste à renvoyer tous les produits réalisables à partir d'un *feature model*.

Calcul du nombre de produits. Cette opération consiste à renvoyer le nombre de produits pouvant être créés en partant du *feature model*. Cette opération peut également servir à vérifier la satisfiabilité du modèle en posant que celui-ci est valide s'il existe au moins un produit. Cette opération est utile en ingénierie car si le nombre de produits augmente, la *SPL* devient plus souple et donc plus complexe. Par exemple, dans le *feature model* illustré sur la figure 1, il y a 15 produits possibles.

Feature morte. Cette opération consiste à détecter une feature morte. Le fait qu'un *feature model* soit satisfiable n'exclut pas la présence d'une feature morte. Ce terme signifie qu'une feature ne pourra jamais être selectionnée quelle que soit la configuration. Des exemples typiques générant une feature morte sont illustrés sur la figure 7. Il y a également d'autre détections d'anomalies possibles telles que la redondance par exemple [10]

Configuration. Cette opération réduit le nombre de produits possibles dans un *feature model* en permettant la sélection ou la désélection

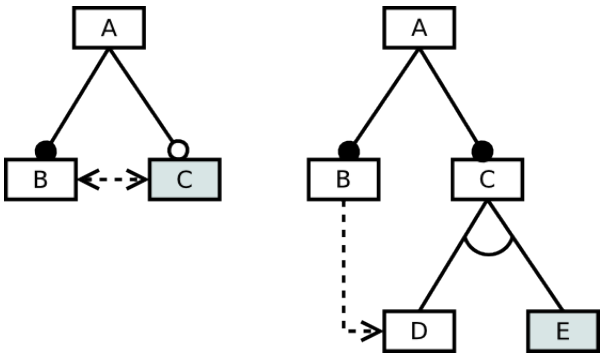


FIGURE 7 – Exemple de feature morte. Celles-ci sont représentées en gris.

de ceux-ci. Le *feature model* évolue en fonction du temps dans le sens que le nombre de produits diminue à chaque étape. Ceci est connu sous le nom de configuration par stade [16](*staged configuration*) ou configuration interactive [9]. Durant celle-ci, l'utilisateur fait des choix pour créer un produit tandis que l'outil guide la configuration en rendant les choix consistants. En fait, le système valide et propage automatiquement les choix de l'utilisateur pour être sûr qu'il puisse à tout moment aboutir à un produit valide [27].

2.1.4 Support permettant l'automatisation

Logique des propositions.

La transformation d'un *feature model* vers la logique des propositions se retrouve sur le tableau 1 [27, 10, 41]. On y trouve également un exemple concret en utilisant le *feature model* illustré sur la figure 1.

Relation	**Description**	**Logique des propositions**	**Exemple**
Root	R est la racine	R	Car
Mandatory	C est une sous-feature obligatoire de P	$P \leftrightarrow C$	$Car \leftrightarrow Body$
Optional	C est une sous-feature optionnel de P	$C \rightarrow P$	$GPS \rightarrow Car$
Alternative	Les $X_1, .., X_n$ sont les sous-features de la relation *alternative* ayant comme parent P	$P \rightarrow$ 1-of-n$(X_1, .., X_n)$	$P \leftrightarrow (Manual \wedge \neg Automatic) \vee (\neg Manual \wedge Automatic)$
Or	Les $X_1, .., X_n$ sont les sous-features de la relation *or* ayant comme parent P	$P \leftrightarrow (X_1 \vee .. \vee X_n)$	$Engine \leftrightarrow (Electric \vee Gasoline)$
Requires	X implique Y	$X \rightarrow Y$	$High \rightarrow GPS$
Excludes	X et Y s'excluent mutuellement	$\neg(X \wedge Y)$	$\neg(Manual \wedge High)$

TABLE 1 – Mapping du *feature model* vers la logique des propositions

En utilisant le tableau 1, on peut transformer le *feature model* présenté sur la figure 1 en logique des propositions :

$$Car \wedge (Car \leftrightarrow Body) \wedge (Car \leftrightarrow Engine) \wedge (Car \leftrightarrow Transmission) \wedge$$
$$(GPS \rightarrow Car) \wedge (Car \leftrightarrow Interior) \wedge$$
$$(Engine \rightarrow (Electric \vee Gasoline)) \wedge$$
$$(Electric \rightarrow Engine) \wedge (Gasoline \rightarrow Engine) \wedge$$
$$(Manual \rightarrow Transmission) \wedge (Automatic \rightarrow Transmission) \wedge$$
$$(Transmission \rightarrow ((Manual \wedge \neg Automatic) \vee (\neg Manual \wedge Automatic))) \wedge$$
$$(Standard \rightarrow Interior) \wedge (High \rightarrow Interior) \wedge$$
$$(Interior \rightarrow ((Standard \wedge \neg High) \vee (\neg Standard \wedge High))) \wedge$$
$$(High \rightarrow GPS) \wedge (\neg(High \wedge Manual))$$

La dernière ligne représente les contraintes *cross-tree*.

La logique de description.
La logique de description (*Description Logic – DL*) est une famille de langages formels de représentation de connaissance. Elle est plus expressive que la logique des propositions. Un problème décrit en *DL* est souvent représenté par un ensemble de concepts (classes), un ensemble de rôles (propriétés ou relations) et un ensemble d'individus (instances) [10].

Wang *et al.*[36] ont été les premiers à proposer une solution basée sur la logique de description. Dans cette étude, ils proposent un ensemble de règles de mappage entre le *feature model* et le langage *OWL DL* [20]. Il existe en fait trois types de langages *OWL* : *Lite*, *DL* et *Full*. L'avantage d'*OWL DL*

est qu'il permet le maximum d'expressivité tout en restant calculable et décidable.

Pour raisonner sur les logiques de descriptions, un moteur d'inférence (*Racer* (*Renamed ABox and Concept Expression Reasoner*) [1], *Pellet*, *Fact++*,...) est nécessaire. Un moteur d'inférence est un programme informatique prenant comme entrée une base de connaissance pour laquelle il essaie de dériver des réponses/conclusions [42]. Il permet, entre autre, la vérification de consistance.

Problèmes de satisfaction de contraintes.

Outre les solutions précédentes, l'emploi de la programmation par contrainte est également possible pour effectuer des analyses automatiques sur un *feature model* [9]. Pour se faire, on convertit le feature model vers un problème de satisfaction de contraintes (*Constraint Satisfaction Problem – CSP*). Un problème *CSP* est un problème mathématique défini comme un ensemble dont l'état doit satisfaire un certain nombre de contraintes ou de limites [39]. Plusieurs études ont été réalisées mais la plus complète semble celle de Benavides *et. al* qui proposent la traduction des features mais également des attributs en *CSP* [8]. On peut trouver un mapping général d'un *feature model* (sans attributs) vers un problème CSP dans l'article de Benavides *et.al* [10]

Différents outils existent pour résoudre des problèmes CSP : *JaCoP*, *Choco*, *OPL Studio*, *GNU Prolog*, ... [10].

1. http ://www.racer-systems.com/

2.1.5 Notations graphiques

En ce qui concerne la représentation graphique de *feature models*, divers outils existent. Cependant, ils restent souvent des prototypes. Chaque outil peut disposer de certaines caractéristiques (partiellement de [33]) :

- une ou plusieurs sortes de visualisation des features (arbre, liste indentée, ...)
- présence des contraintes ou des attributs
- visualisation graphique ou textuelle des contraintes et des attributs
- emploi d'un langage textuel pour représenter le *feature model* (*XML*, *GUISL*, ...)
- utilisation d'un ou plusieurs solveurs (*SAT*, *BDD*, *CSP*, ...)
- automatisations possibles (Satisfiabilité du modèle, calculer le nombre de produits, ...)

Voici une liste non-exhaustive de différents outils [41, 33] :

- Ahead Tool Suite — `http://www.cs.utexas.edu/users/schwartz/ATS.html`
- FaMa Tool Suite — `http://www.isa.us.es/fama/`
- Pure : :Variants — `http://www.pure-systems.com/Variant_Management.49.0.html`
- Feature Modeling Plug-in (Plug-in Eclipse) — `http://gsd.uwaterloo.ca/fmp`
- Feature Modeling Tool (Plug-in Visual Studio) — `http://giro.infor.uva.es/FeatureTool.html`
- FeatureIDE — `http://www.cs.utexas.edu/users/schwartz/ATS.html`
- SPLOT (Software Product Line Online Tools) `http://www.splot-research.org/`

- XFeature — `http://www.pnp-software.com/XFeature/`

On peut trouver une comparaison de différents outils de visualisation dans la thèse de Saiz [33].

2.1.6 Notations textuelles

Au-delà des représentations graphiques, il existe également des représentations textuelles pour les *feature models*. Un de leurs avantages est qu'elles ne nécessitent pas d'outils spécifiques pour leur création et un simple éditeur de texte (*Notepad++*, *Gedit*, ...) suffit. Ci-dessous, se trouve une liste non-exhaustive de langages textuels :

- **TVL** (*Textual Variability Langage*) sera détaillé en section 3.1.
- **FDL** est le premier langage textuel pour représenter des *feature models*. Il ne prend en compte ni les attributs, ni les cardinalités et autres structures plus avancées. [13]
- **XML** s'utilise également pour representer un *feature model* et se voit utilisé par des outils tels que *Feature Modeling Plug-in*, *FeatureIDE* (v2.6), *Pure : :Variants* ou encore *FaMa Tool Suite*. *XML*, dans notre cas, n'est généralement pas destiné à une utilisation manuelle (lecture ou écriture) étant donné son aspect relativement verbeux et le fait que chaque outil possède sa propre syntaxe.
- **GUIDSL** est une grammaire pour représenter les *feature models* proposée par Batori [6]. Elle est employée par des outils tels que *Ahead* ou *FeatureIDE* (v2.5). Malheureusement, elle est assez limitée car elle ne prend pas en compte les attributs, les cardinalités et la visualisation hiérarchique des features [13].
- **SXFM** est une syntaxe utilisée par *SPLOT* par exemple. Ce format

utilise XML pour les métadonnées ainsi que pour la structure, mais il se veut lisible par l'être humain. Contrairement à *GUIDSL*, il montre explicitement la hiérarchie dans un *feature model* grâce à l'indentation.

- **Clafer** (class,feature, reference) est un langage de modélisation textuel qui fut proposé par Bak *et al.* [3]. Il est approprié pour la modélisation de domaines et pour la capture de connaissances. Cette notation est vraiment concise et peut être employée pour représenter des métamodèles et des *feature models* ou une combinaison des deux.
  Pour obtenir un complément d'information au sujet de ce langage, on peut visiter le tutorial proposé par Antkiewicz [2]. Un exemple de feature model défini avec le langage *Clafer* est présenté en figure 1.

Listing 1 – Exemple en *Clafer*

```
1  Car
2   [~(High && Manual)]
3    Body
4    or Engine
5       Electric
6       Gasoline
7    xor Transmission
8       Manual
9       Automatic
10   GPS?
11   xor Interior
12      Standard
13      High
```

```
14          [GPS]
```

Pour plus de détails, j'invite le lecteur à lire l'article de Classen *et al.* [13] où divers langages textuels de *feature models* sont présentés brièvement et ensuite comparés entre eux.

2.2 Satisfiabilité

En informatique, la satisfiabilité est le fait de savoir si les variables d'une formule booléenne peuvent être assignées pour que l'on rende la formule vraie *true*[2]. Si une attribution est possible, la formule est dite satisfiable (*satisfiable*), dans le cas contraire, insatisfiable (*unsatisfiable*). Par exemple, la proposition $(p \wedge q)$ sera satisfiable lorsque [p=true, q=true]. La proposition $(p \wedge q \wedge \neg p)$, quant à elle, sera insatisfiable étant donné qu'aucune valeur ne peut être attribuée à p [27, 38].

La satisfiabilité booléenne est un problème de calcul assez complexe. Il fait partie de la classe de complexité NP-complet (NPC). Ce type de problème a deux propriétés :

- La classe NP-complet est un sous-ensemble de NP, l'ensemble des problèmes de décision où il est possible de vérifier une solution en temps polynomial sur une machine de Turing non-déterministe.
- Un problème p, appartenant à l'ensemble NP, fait partie de NPC, si et seulement si, tous les autres problèmes de NP peuvent être transformés en p en un temps polynomial. Cela signifie que le problème p est au moins aussi difficile que tous les autres problèmes de classe NP.

2. `http://en.wikipedia.org/wiki/SAT_solver`

Bien que vérifier une solution d'un problème NPC est relativement rapide, la trouver n'est pas efficace. [43]

2.2.1 Vérification de la satisfiabilité

Pour vérifier la satisfiabilité de modèles en utilisant la logique des propositions, plusieurs solutions sont possibles. En voici la description de deux : Solveur SAT, Solveur *BDD*.

Solveur SAT. Pour vérifier la satisfiabilité d'une formule propositionnelle, il existe ce que l'on appelle des solveurs SAT. La plupart d'entre eux utilisent, comme procédure de décision principale, l'algorithme DPLL. Cet algorithme permet de résoudre efficacement des problèmes larges. Les formules proposées aux solveurs SAT sont souvent en forme normale conjonctive (*conjunctive normal form (CNF)*).

Une formule *CNF* consiste en une conjonction de clauses où chaque clause est une disjonction de littéraux où un littéral représente une variable ou sa négation : $C_1 \vee .. \vee Cn$. Il arrive également que l'on écrive une formule *CNF* sous la forme d'un ensemble $\{C_1, .., C_n\}$ ou que l'on remplace simplement les $\wedge$ par des virgules $(C1, .., C_n)$. Voici un exemple de formule *CNF* contenant quatre variables $a,\ b,\ c,\ d$: $(A \vee \neg B) \wedge (C \vee D) \wedge (A \vee D)$. Cette formule contient trois clauses et chacune d'elle est composée de deux littéraux.

DPLL (*Davis-Putnam-Logemann-Loveland*) [40, 11, 30] est un algorithme de recherche basé sur le backtracking qui vit le jour en 1962. L'algorithme basique fonctionne comme suit : on choisit un littéral et on lui attribue

une valeur de vérité (*true* ou *false*) ; ensuite, on simplifie la formule et l'on vérifie récursivement si la formule simplifiée est satisfiable ou non. Si elle est consistante, la formule originale est satisfiable, sinon, on recommence la vérification en utilisant la valeur de vérité opposée. La simplification enlève toutes les clauses qui deviennent vraies à l'attribution et tous les littéraux qui deviennent faux dans les clauses restantes.
L'algorithme DPLL améliore l'algorithme de backtracking de base en y ajoutant des règles supplémentaires à chaque étape :

- Unit Propagation Rule : Pour chaque clause (l) appelée une clause unitaire, on supprime toutes les clauses contenant l et tous les littéraux $\neg l$.
- Pure Literal Rule : Si un littéral l apparaît dans quelques clauses mais que $\neg l$ n'apparaît jamais, alors on peut supprimer toutes les clauses contenant l. Le littéral l est appelé un littéral pur (*pure literal*). Etant donné qu'il n'existe qu'avec une seule polarité, on peut, dans tous les cas, lui assigner une valeur pour qu'il rende toutes les clauses le contenant vraies.

En se basant sur la formule *CNF* suivante $(\neg a \vee \neg b) \wedge (b \vee c) \wedge (\neg a \vee \neg c \vee d) \wedge (b \vee \neg c \vee \neg d) \wedge (a \vee d)$ [30], les différents appels de l'algorithme DPLL sont présentés sur la figure 8. Comme on peut le constater, toutes les branches de l'arbre n'ont pas été parcourues (dénotées par ?). En réalité, le choix d'un littéral à chaque étape a un impact sur l'efficacité de l'algorithme. Par conséquent, on peut arriver plus rapidement à une solution en partant d'un littéral plutôt qu'un autre [40].

Actuellement, les solveurs SAT sont basés sur différentes variations de

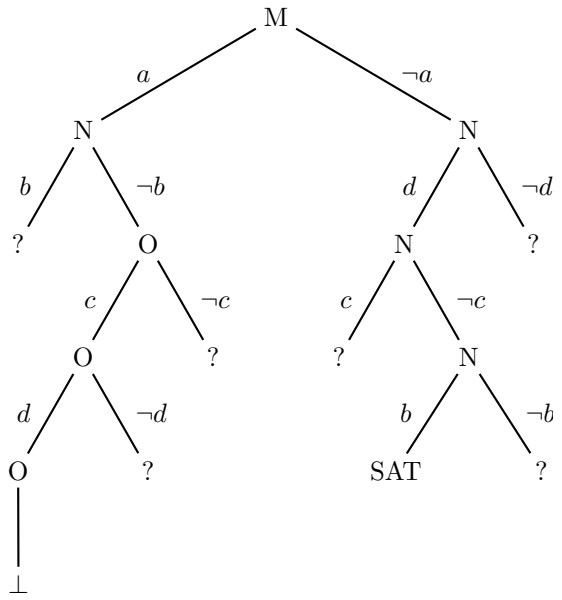


FIGURE 8 – Arbre d'appel DPLL

la procédure de l'algorithme DPLL. Les améliorations importantes de performances sont dues à des meilleures techniques d'implémentation et à plusieurs perfectionnements conceptuels du DPLL original ayant comme but la réduction de l'espace de recherche comme, par exemple, le *backjumping* (une forme de backtracking non-chronologique). Ces avancées permettent de vérifier la satisfiabilité de problèmes SAT industriels avec des dizaines de milliers de variables et des millions de clauses [30].

Binary Decision Diagram. Il existe également, pour vérifier la satisfiabilité d'une formule propositionnelle, des solveurs *Binary Decision Diagram* (BDD) qui prennent en argument une formule logique comme entrée (pas forcément en *CNF*) et qui la représentent par la suite sous forme de graphe permettant de déterminer si la formule est satisfiable ou non.[10].

L'emploi de *BDD* peut être utile pour résoudre quelques automatisations parfois difficiles pour un solveur SAT. Par exemple, le calcul du nombre de produits possibles ou la rétro-ingénierie de modèle [27, 10].

Un *BDD* est donc un graphe dirigé acyclique (*directed acyclic graph* (*DAG*)) avec un ou deux nœuds terminaux qui ont un degré sortant nul et qui sont étiquetés 0 ou 1. Il est également composé d'un ensemble de nœuds variables ayant un degré extérieur égal à deux. Chaque nœud est représenté par une variable booléenne et ses enfants sont nommés bas (*low*) et haut (*high*). L'arc sortant vers un fils représente la valeur assignée au nœud parent qui peut être soit 0 pour le bas, soit 1 pour le haut. Sur la figure suivante, ils sont indiqués par une ligne pointillée et par une ligne pleine respectivement. Un *BDD* peut également être ordonné (*Ordered — OBDD*) si tous les chemins à travers le graphe respectent un ordre linéaire $x_1 < .. < x_n$. Un (O)BDD peut égalment être réduit (*Reduced* — R(O)BDD). [1, 37]

Sur la figure 9, un exemple de BDD représentant $(x_1 \leftrightarrow y_1) \wedge (x_2 \leftrightarrow y_2)$ est illustré. Cet exemple est également un *OBDD* suivant l'ordre $x_1 < y_1 < x_2 < y_2$. [1]

2.3 SMT

Satisfiability Modulo Theories (SMT) est un domaine de la déduction automatique qui étudie des méthodes pour vérifier la satisfiabilité de formules logiques en tenant compte de la composition de certaines théories logiques τ. On peut citer, par exemple, la théorie des entiers (*Integers*) ou des réels (*Reals*) ainsi que des théories de structure de données tels que les listes (*Lists*), les tableaux (*Arrays*), les vecteurs de bits (*Bitvectors*), ... On

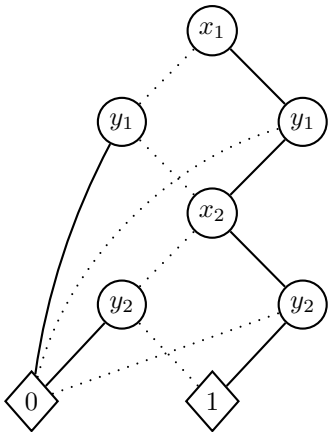


FIGURE 9 – Exemple de BDD

expliquera certaines d'entre elles en davantage de détails dans la section 3.2.1 (p50). [4, 44]

Les techniques courantes pour trouver la satisfiabilité d'une formule F en tenant compte de théories τ peuvent globalement être divisées en deux catégories principales : *eager* et *lazy* [30, 44].

Eager. La formule d'entrée *SMT* est convertie grâce à l'utilisation d'une transformation préservant la satisfiabilité en une formule *CNF* qui est ensuite passée dans un solveur SAT. Une des forces de cette approche est qu'elle permet l'emploi systématique des meilleurs solveurs SAT disponibles. Cependant, cette technique n'est pas très flexible [30]. En effet, pour être efficace, des techniques sophistiquées de traduction sont requises pour chaque théorie. De plus, cela signifie que le solveur SAT doit travailler plus que nécessaire pour trouver des faits "évidents" (comme par exemple, $x + y = y + x$, pour une addition d'entiers) [44].

Lazy. Une alternative à la technique précédente est l'utilisation d'un sol-

veur τ spécifique pour décider de la satisfiabilité des conjonctions de prédicats d'une théorie particulière. Un avantage de la théorie *lazy* est qu'elle est flexible car elle peut combiner n'importe quel solveur SAT avec n'importe quel solveur τ (τ-*Solver*).

2.3.1 Solveurs *SMT*

Voici une liste alphabétique non-exhaustive des différents solveurs *SMT* [14, 44]. Un solveur peut supporter plusieurs langages d'entrée. Par conséquent, j'ai décidé de selectionner uniquement les solveurs supportant (partiellement) le langage *SMT-LIBv2*.

- CVC3 — `http://www.cs.nyu.edu/acsys/cvc3/`
- OpenSMT — `http://www.verify.inf.unisi.ch/opensmt`
- MiniSMT — `http://cl-informatik.uibk.ac.at/software/minismt/`
- MathSAT — `http://mathsat4.disi.unitn.it/`
- SONOLAR — `http://www.informatik.uni-bremen.de/~florian/sonolar/`
- STP — `http://sites.google.com/site/stpfastprover/`
- Z3 — `http://research.microsoft.com/en-us/um/redmond/projects/z3/index.html`

Comme mentionné précédemment, il existe plusieurs théories dans le domaine *SMT* et chaque solveur ne supporte pas forcément toutes les théories.

En ce qui concerne la performance des solveurs *SMT*, il existe une compétition depuis 2005 : *SMT-COMP*[3]. Sur le site, on peut retrouver un comparatif des solveurs en fonction des différentes logiques.

3. http ://www.smtcomp.org/

2.3.2 Langages

Voici une liste non-exhaustive des divers langages existants pour représenter des formules *SMT*.

- **SMT-LIB** (*Satisfiability Modulo Theories Library*) est une initiative visant principalement à développer un langage d'entrée et sortie pour les solveurs SMT. Il est détaillé dans la section 3.2.
- **CVC** peut être utilisé avec les solveurs *CVC3* et *STP*.
- **STP** peut être utilisé avec les solveurs *STP*. C'est un langage qui reprend la plupart des fonctions et prédicats implémentés dans un langage tel que *C* ou dans un jeu d'instruction machine, si ce n'est qu'il ne présente pas le type de données flottantes et les opérations sur ce dernier. [22]. Par l'intermédiaire du solveur *STP*, on peut, par exemple, convertir le langage d'entrée vers le langage *SMT-LIB* ou *CVC*.
- **Simplify** peut être utilisé avec *z3*, par exemple, mais il est désormais déprécié et les auteurs de *z3* conseillent d'employer *SMT-LIB*[4].

4. http ://research.microsoft.com/en-us/um/redmond/projects/z3/group__simplify.html

3 Background

Dans cette section, on traite les éléments principaux de la problématique en définissant les deux langages utilisés : TVL et SMT-LIBv2.

3.1 TVL

Cette sous-section se base sur les documents expliquant la syntaxe et la sémantique du langage TVL [12] [13].

TVL est un langage textuel de modélisation de features. C'est un langage léger et compréhensible. Il couvre aussi bien la notation classique introduite par Kang *et al.* [24] que la décomposition basée sur les cardinalités et les attributs. Son objectif principal est de fournir un langage lisible par l'homme et de supporter des modèles à grande échelle grâce à des mécanismes de modularisation. En effet, comme la plupart des notations sont graphiques, travailler avec des *feature diagrams* de grandes dimensions peut très vite devenir fastidieux. *TVL* se définit par une grammaire *LALR*, une sémantique formelle et une implémentation de référence qui est disponible en ligne [5].

TVL utilise une syntaxe du même style que celle du langage *C* : il utilise des accolades pour délimiter les blocs, un point-virgule pour terminer une assertion et la même syntaxe pour les commentaires (`//` ou `/*..*/`). En ce qui concerne la déclaration de features et d'attributs, ils doivent commencer par une majuscule et une minuscule respectivement.

Pour plus de détails sur la syntaxe et la sémantique de *TVL*, j'invite le

5. http ://www.info.fundp.ac.be/∼acs/tvl

lecteur à lire l'article de Classen *et al.* [12] [13].

3.1.1 Décomposition

En partant du *feature diagram* présenté en section 2.1 sur la figure 1, on obtient le modèle *TVL* suivant :

Listing 2 – Exemple en *TVL*

```
1  root Car {
2      High requires GPS;
3      High excludes Manual;
4
5      group allOf {
6          Body,
7          Engine {
8              group someOf {
9                  Electric,
10                 Gasoline
11             }
12         },
13         Transmission {
14             group oneOf {
15                 opt Manual,
16                 Automatic
17             }
18         },
19         opt GPS,
20         Interior {
```

```
21              group oneOf {
22                  Standard,
23                  High
24              }
25          }
26      }
27  }
```

Un type de décomposition est défini en TVL avec le mot-clé **group**. Les différents opérateurs sont :

- **allOf** qui représente une décomposition *and*
- **oneOf** qui représente une décomposition *xor*
- **someOf** qui représente une décomposition *or*
- **group [i..j]** qui permet de spécifier une décomposition basée sur les cardinalités, où i et j sont ses bornes inférieure et supérieure. Lors de l'utilisation de cette décomposition, on peut employer le caractère astérisque (*) pour signifier le nombre de sous-features comprises dans le groupe. Par exemple, l'utilisation de **group [1..*]** est équivalente à **group someOf**.

Une feature optionnelle sera définie, quant à elle, avec le mot-clé **opt**. Dans notre exemple (fig. 1), la feature **GPS** est optionnelle.

3.1.2 Attributs

TVL donne également la possibilité de définir des attributs à une feature. Quatre types d'attributs sont supportés : entier (**int**), réel (**real**), booléen (**Bool**) et énumération (**enum**). Pour déclarer un attribut, il suffit de définir

son type et son nom dans le bloc de la feature auquel il appartient.

Lors de l'emploi d'un attribut de type **enum**, celui-ci prend une seule valeur parmi l'ensemble des valeurs prédéfinies. Notons aussi que dans ce cas le mot-clé **in** est obligatoire. En *TVL*, les crochets servent à définir un intervalle contrairement aux accolades qui décrivent une liste. Différentes manières de déclarer un attribut sont présentées sur la figure 10 [13].

`Feature { int x; }`	`Feature { int x in [0..5]; }`
(a) Basic	(b) Restriction de domaine
`Feature { int x is 5; }`	`Feature { int x, ifIn : is 10, ifOut : is 0; }`
(c) Valeur fixe	(d) Valeur conditionnelle

FIGURE 10 – Déclaration d'attributs en *TVL*

En partant de notre exemple (fig. 1), on peut imaginer qu'un GPS a un attribut résolution (**resolution**) de type énuméré. Il peut donc avoir une valeur parmi l'ensemble {*360p*, *480p*, *720p*}. Le GPS peut également avoir une mémoire (**memory**) allant de 32 à 512 Mo. En TVL, on obtient :

```
GPS {
    enum resolution in {360p, 480p, 720p};
    int memory in [32..512]
}
```

3.1.3 Expressions

On peut utiliser différentes expressions en *TVL* pour soit exprimer la valeur d'un attribut, soit exprimer une contrainte. Les expressions peuvent être de types booléen (*bool*), entier (*integer*) ou réel (*real*).
Ces expressions peuvent être combinées avec les opérateurs classiques. Pour les entiers, il s'agit de $+, -, /, *, abs$. Pour les opérateurs booléens, il s'agit de $!, \&\&, ||, ->, <-, <->$. Les expressions booléennes peuvent également être combinées avec des comparaisons d'entiers ou réels (étant donné qu'elles renvoient un booléen) : $<, <=, >=, >$. Il existe également des fonctions d'aggrégation : $sum, mul, min, max, avg, count, or, xor$. Elles prennent une liste d'expressions comme argument.

Pour les contraintes de base, il existe également les mots-clés *requires* et *excludes*. En revanche, ils ne peuvent être utilisés qu'avec une feature de part et d'autre du mot clé. La ligne 1 est syntaxiquement correcte contrairement à la ligne 2. Pour contourner cela, on peut employer des expressions booléennes classiques (ligne 3 ou 4 représentent la même chose).

```
1 High excludes Manual; // Correct
2 High requires and(GPS, Automatic); // Incorrect
3 High -> and(GPS, Automatic); // Correct
4 High -> (GPS && Automatic); // Correct
```

3.1.4 Contraintes

Comme expliqué dans la section 2.1, un *feature model* peut posséder des contraintes *cross-tree* entre ses features. Les contraintes sont des expressions booléennes et peuvent être ajoutées dans le bloc d'une feature particulière en compagnie de la déclaration d'attributs.
En *TVL*, il existe également un sucre syntaxique pour les contraintes gardées. Elles permettent de spécifier une valeur d'un attribut dans le cas où sa feature parent est selectionnée (**ifIn :**) ou non (**ifOut :**). Dans l'exemple illustré sur la figure 10), x vaut 10 si la feature est selectionnée, sinon, 0.

3.1.5 Modularisation

Comme énoncé précedemment, un des principaux buts de TVL est d'aider les différents utilisateurs à développer des modèles de grande dimension. Le premier moyen est d'inclure un fichier grâce à la commande **include** qui prend en argument le chemin vers le fichier que l'on veut inclure. Cette commande inclut le fichier au point où elle a été appelée.

```
include(./some/other/file);
```

Un autre mécanisme consiste à déclarer des nouveaux types personnalisés. Ils sont définis au début du fichier *TVL* et peuvent, par la suite, être réutilisés. Dans cet exemple, on définit une liste de langues qui sera réutilisée plus tard sur un attribut de la feature *GPS*.

```
enum languageList in {French, English, Dutch,
    German};
...
GPS {
```

```
    languageList language;
}
```

Il est également possible de déclarer des constantes (**const**) avec un type, un nom et une valeur. On peut ensuite les utiliser dans des expressions ou des cardinalités.

```
const int maxSpeed 250;
```

TVL laisse également le choix à l'utilisateur d'organiser son *feature model* comme il le désire. En effet, il peut d'abord déclarer la feature *GPS* faisant partie de *Car* et ensuite développer *GPS*. Il peut, par exemple, lui ajouter un attribut comme dans l'illustration suivante (Listing 3.1.5).

```
root Car {
  ..
  opt GPS
  ..
}

GPS {
   enum resolution in {360p, 480p, 720p};
}
```

3.2 SMT-LIB

Cette section se base sur les documents officiels concernant SMT-LIBv2.[4, 5, 14]

SMT-LIB est l'abréviation pour *The Satisfiability Modulo Theories Library*. *SMT-LIB* est une initiative internationale visant à faciliter la recherche et le développement dans le domaine *SMT*. Elle a différents buts :

- fournir des descriptions claires et standardisées des théories utilisées dans les systèmes *SMT* ;
- développer et promouvoir un langage d'entrée et de sortie pour les différents solveurs *SMT* ;
- établir et mettre à disposition de la communauté une large librairie de benchmarks pour les solveurs *SMT*.

3.2.1 Théories

Voici donc les différentes théories définies par *SMT-LIBv2*.

- Théorie des entiers (*int*)
- Théorie des nombres réels (*real*)
- Théorie des vecteurs de bits (*bitvector*)
- Théorie des tableaux (*array*)
- Théorie des entiers et réels

Il faut noter que chaque théorie contient implicitement la théorie cœur *Core*.

Il est uniquement question, ici, des théories utilisées dans le cadre de cette étude en définissant la théorie cœur et la théorie des vecteurs de bits.

Théorie cœur. Celle-ci définit les éléments basiques de la logique booléenne.

- Le type *Bool* (booléen) ;
- Les constantes *true* et $false$ de type *Bool* ;
- L'opération *not* ;
- Les fonctions de base *and*, *or* et *xor* associative à gauche sur les

valeurs booléennes ;

- La fonction => correspondant à l'implication associative à droite.
- La fonction égalité entre des éléments d'ensembles de valeur de même type.
- La fonction inégalité entre des éléments d'ensembles de valeur de même type
- La fonction "if-then-else" qui prend comme premier argument un booléen et 2 arguments supplémentaires d'un même type.

Théorie des vecteurs de bit. Celle-ci définit le comportement des vecteurs de bits. On définit un type différent de vecteur de bits en fonction de la longueur du vecteur. Des opérations sont définies pour manipuler, combiner et extraire des portions de vecteurs de bits. Certaines de ces opérations interprètent le vecteur de bits comme un nombre naturel. Le vecteur de bits est un ensemble de binaires non-signés qui représente les entiers positifs avec le bit le moins important à droite.

- Un type (_ Bitvec n) est défini pour chaque valeur n non-nulle.
- Les littéraux binaires et hexadécimaux sont définis pour avoir un type de vecteur de bits correspondant à leur longueur.
- La fonction *concat* permet de combiner deux vecteurs de bit en un seul.
- Les différentes fonctions (*extract* i j) sont définies pour extraire une portion continue du vecteur de bits de l'index i à j (compris)
- Les fonctions unaires *bvnot* et *bvneg*
- Les fonctions binaires *bvand bvor bvadd bvmul bvudiv bvurem bvshl bvlshr*
- La fonction de comparaison binaire *bvult*

3.2.2 Logiques

Une logique consiste en une ou plusieurs théories ainsi que des restrictions sur les sortes d'expressions utilisées dans cette logique. [14]

Logique booléenne. `QF_UF`. Cette logique incorpore seulement la théorie cœur fournissant le type *bool* et les différentes opérations classiques sur les booléens. L'utilisateur peut ajouter des types supplémentaires et des fonctions non-interprétées. En revanche, aucun quantificateur n'est autorisé dans une expression.

Logique arithmétique. Les différentes logiques sont `QF_LIA, QF_NIA, QF_LRA, QF_AUFLIA, AUFLIA, AUFLIRA, AUFNIRA, LRA`.

Logique de différence arithmétique. La différence arithmétique autorise seulement une comparaison entre deux valeurs numériques ou une comparaison entre la différence de valeurs numériques et un littéral numérique positif ou négatif. Les différentes logigues sont `QF_IDL, QF_RDL, QF_UFIDL`.

Logique avec des vecteurs de bits et des tableaux Les différentes logiques sont `QF_BV, QF_AX, QF_ABV, QF_AUFBV`.

3.2.3 Expression

L'application d'une fonction sur une séquence d'argument est de la forme :

$$(< qualified - identifier >< expr > +)$$

La syntaxe *SMT-LIB* a un style *prefix* (les opérateurs sont écrit avant les opérandes). La notation *infix* n'est jamais utilisée, même pour de l'arithmétique conventionnelle. En SMT-LIB, nous écrivons (+ 1 2) et non (1 + 2).

L'encadré suivant correspond à la déclaration d'une fonction. *symbol* y représente le nom de la fonction et le $< sort - expr >$ (droite) permet de définir son type. Le terme $(< sort - expr > *)$ représente les arguments que la fonction *symbol* peut prendre. Cependant, ce terme vaut toujours () dans cette analyse étant donné que nous utilisons des fonctions sans arguments.

$$(declare - fun < symbol > (< sort - expr > *) < sort - expr >)$$

Etant donné que l'on a fait appel aux termes associativité à gauche et à droite, voici un rappel de ces principes :

- Associativité à gauche : *or* est une fonction définie dans la théorie cœur qui prend deux arguments booléens (*bool*), mais peut également être appelée sur plusieurs arguments en utilisant l'associativité à gauche. Par conséquent, $(or\ a\ b\ c\ d)$ est équivalent à $(or\ (or\ (or\ a\ b)\ c)\ d)$. D'autres fonctions ont le même comportement telles que les opérations booléennes *xor* et *and* ainsi que les opérations sur les entiers et les réels.
- Associativité à droite : L'implication (=>) est, quant à elle, associative à droite. L'expression $(=>\ a\ b\ c\ d)$ est équivalente à $(=>\ a\ (=>\ b\ (=>\ c\ d)))$

Note : Avec le solveur SMT z3, on peut aussi appeler l'implication avec *implies*. Par exemple, dans le listing 3.2.3, ces deux lignes sont équivalentes.

```
1 (assert (implies a b))
2 (assert (=> a b))
```

3.2.4 Constantes

Si la logique comprend les bitvectors, les constantes suivantes sont réalisables.

#bX ou X est un nombre binaire de taille m définit un *bitvector* constant de valeur X et de taille m.

Par exemple : #b0001 définit le bitvector de valeur 0001 et de taille 4.

#xX ou X est un nombre hexadécimal de taille m et définit un *bitvector* constant de valeur X et de taille $4 * m$ (étant donné qu'un chiffre hexadécimal est représenté par 4 bits)

Par exemple : #xFE définit le bitvector de valeur FE et de taille 8.

Lors de l'emploi de *bitvectors*, il est également possible de les définir avec cette commande où X représente une valeur numérique (en base 10) et m représente le nombres de bits du vecteur.

$$(\_ bvX\ m\ )$$

Dans l'encadré suivant, les expressions à gauche et à droite sont équivalentes. Elles représentent chacune un bitvecteur de taille 5 représentant le chiffre 7.

$$\text{\#b00111} \equiv (\_\ \text{bv7}\ 5\ )$$

Note : Pour le moment, le solveur *SMT z3* n'accepte pas la notation **#bX** mais accepte l'autre. *STP* lui, les accepte.

3.2.5 Logique *QF_BV*

Utilisant initiallement le solveur *STP* [22], j'ai décidé de me concentrer sur une solution basée sur la logique *QF_BV* [31, 14]. Pour rappel, cette logique permet l'utilisation de vecteurs de bits (*bitvectors*). Par conséquent, la représentation d'un attribut se fera à l'aide d'un vecteur de bits.

Cette logique permet l'emploi d'expressions sans quantificateur. Elle inclut la famille des différents types de *bitvectors* et les fonctions définies dans la théorie des vecteurs de bits *bitvectors*.

En règle générale, lorsqu'une fonction prend deux *bitvectors* en entrée, ceux-ci doivent être de la même taille.

La commande ci-dessous définit la variable x comme un *bitvector* comportant m bits.

```
( declare-fun x () (_ BitVec m) )
```

Comparaison sur les *bitvectors*

Voici différents prédicats binaires permettant de comparer deux *bitvectors*. Il existe différents prédicats en fonction de la représentation du nombre binaire qui peut être soit signé (*signed*), soit non-signé (*unsigned*). Pour rappel, lorsqu'un nombre binaire est signé, le premier bit (à gauche) correspond au bit du signe. Les nombre positifs sont représentés normalement, mais lorsqu'il s'agit d'un nombre négatif, on utilise la notation en complément à deux. Par exemple, en nombre binaire signé (sur 8 bits) : -5 vaut *11111011* et 5 vaut *00000101*. La syntaxe pour les différents prédicats est :

```
(predicat (_ BitVec A) (_ BitVec B))
```

Dans le tableau 2, on explique les différentes fonctions de comparaisons sur les *bitvectors*.

Par exemple,

		Non-signé	**Signé**	**Resultat**
A plus petit que B	$A < B$	bvult	bvslt	Bool
A plus petit ou égal à B	$A \leq B$	bvule	bvsle	Bool
A plus grand que B	$A > B$	bvugt	bvsgt	Bool
A plus grand ou égal à B	$A \geq B$	bvuge	bvsge	Bool

TABLE 2 – Prédicats binaires de comparaison

```
(bvuge (_ bv10 3) (_ bv9 3)) donne false
```

3.2.6 Commandes

Voici la liste des commandes [4] principales du langage *SMT-LIB* utilisées :

set-logic L permet de définir la logique que l'on utilise. L'argument L est le nom soit d'une logique appartenant au cataglogue *SMT-LIB*, soit d'une logique d'un solveur particulier. Cette commande doit précéder toutes les autres à l'exception des suivantes : **set-info**, **get-info**, **set-option**, **get-option**, **exit**.

```
( set-logic < L > )
```

check-sat demande au solveur de vérifier si oui ou non la conjonction des différentes formules est satisfiable dans la logique spécifiée par la commande **set-logic**. La réponse à cette commande sera *sat*, *unsat*, *unknow*.

```
( check-sat )
```

- sat : l'ensemble des assertions est satisfiable. Il existe au moins une solution permettant d'assigner une valeur aux différentes constantes et symboles qui rend toutes les assertions vraies.

- unsat : l'ensemble des assertions est insatisfiable. Aucune attribution n'est possible.
- unknow : le solveur ne peut déterminer si l'ensemble des assertions est satisfiable ou non. Plusieurs explications sont possibles. Le solveur peut être à court de mémoire ou de temps. Le solveur peut également avoir trouvé une solution permettant à toutes les assertions d'être vraies, mais il ne peut pas en être sûr à cause de la présence d'assertions quantifiées.

assert t ajoute le terme t à l'ensemble des assertions si t est une formule de forme fermée (*closed formula*). Pour rappel, une formule fermée est une expression qui s'obtient par une combinaison de nombres ou de fonctions et d'opérations de référence [6].

(assert < *expression* >)

exit donne l'instruction au solveur de quitter.

(exit)

Pour plus d'informations concernant les différentes commandes, j'invite le lecteur à lire ces documents officiels [4, 14].

6. http ://fr.wikipedia.org/wiki/Solution_de_forme_fermée

4 Etude de cas

Afin de faciliter la compréhension de la traduction, on met en place un exemple d'une modélisation d'une tablette PC. Celui-ci tentera de capturer un maximum de variabilité et contiendra les différentes structures fournies par *TVL*.

4.1 Description

Notre tablette PC est donc composé de :

- Ecran
- Processeur
- Mémoire
- Camera
- Système d'exploitation
- Clavier
- Applications

L'écran est evidemment obligatoire. Il a une taille (*size*) comprise entre 6' et 10' et il possède également une résolution (*resolution*) qui peut prendre diverses valeurs : `360p`, `480p`, `720p`, 1080p.

Le processeur peut être, quant à lui, composé d'un cœur simple ou double .

La mémoire est interne ou externe. La mémoire interne est obligatoire car elle est nécessaire au fonctionnement de la tablette PC. Il y a la possibilité d'ajouter une ou plusieurs mémoires externes. Ces mémoires externes sont de type *Secure Digital*, *Memory Stick* ou *CompactFlash*.

Une camera au minimum sera présente sur la tablette PC. Elle peut donc

s'y placer à l'avant ou à l'arrière de l'appareil.

Le clavier est optionnel et ne se trouve donc pas systématiquement dans la configuration.

Des applications sont disponibles lors de la composition de l'appareil. Il y'en a quatre : *AngryBirds*, *Facebook*, *Shazaam*, *Skype*.

4.2 Contraintes

Différentes contraintes sont également observables dans notre modèle :

- Si le système est composé du système d'exploitation *iOS*, alors la tablette PC devra être munie d'une caméra à l'avant et à l'arrière.
- On ne peut pas avoir, sur la même tablette, des mémoires externes de type *Memory Stick* et *Compact Flash*.
- Le nombre d'applications choisies doit être entre 2 et 3. Par exemple, des choix possibles seraient : (*AngryBirds*, *Facebook*, *Skype*), (*AngryBirds*, *Shazaam*), ...

4.3 *Feature Diagram*

Un *feature diagram* a pour objectif de représenter graphiquement (sous forme d'arbre) un *feature model*. Le diagramme est ici accompagné de contraintes supplémentaires. Pour rappel, le *feature diagram* permet de voir facilement (étant donné son aspect graphique) les caractéristiques d'une ligne de produits (*SPL*) ainsi que les différentes dépendances qui y figurent. Il faut noter que vu l'emploi de cette notation de *feature model* classique, les différents attributs (par exemple : *int size* pour l'écran) ne sont pas representés. Il en va de même pour les cardinalités du groupe applications (*Applications*).

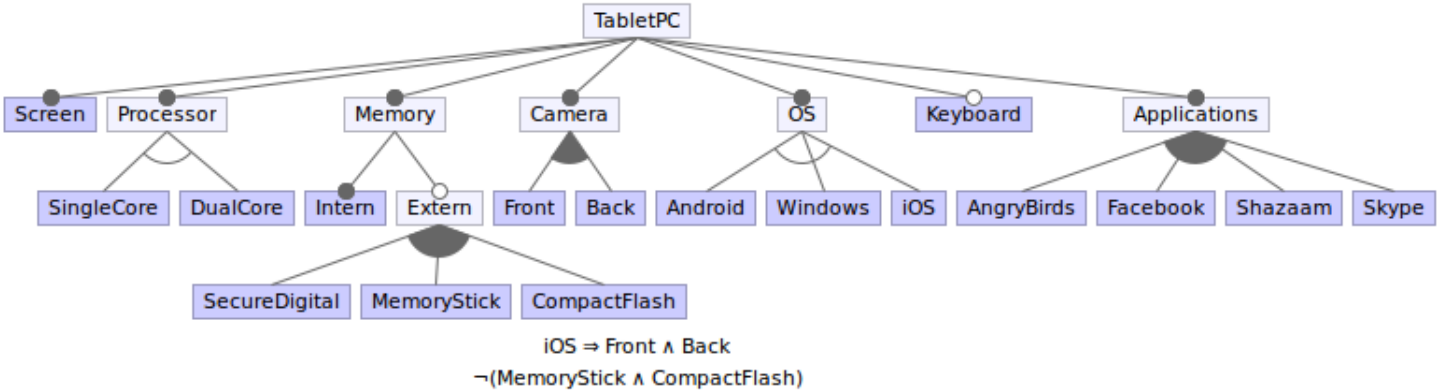


FIGURE 11 – Etude de cas : Feature model

4.4 Code source *TVL*

Ci-dessous, vous trouverez le modèle *TVL* représentant notre tablette PC.

Listing 3 – Etude de cas en *TVL* : Tablette PC

```
1  root TabletPC {
2      IOS -> (Front && Back);
3      MemoryStick excludes CompactFlash;
4      group allOf {
5          Screen {
6            int size in [6..10];
7            enum resolution in {360p, 480p, 720p,
                 1080p};
8          },
9          Processor,
10         Memory,
11         Camera,
12         OS,
13         opt Keyboard,
14         Applications
15     }
16 }
17
18 Memory {
19     group allOf {
20         Intern,
21         Extern
```

```
22      }
23  }
24
25  Extern {
26      group someOf {
27          SecureDigital,
28          MemoryStick,
29          CompactFlash
30      }
31  }
32
33  Camera {
34      group someOf {
35          Front,
36          Back
37      }
38  }
39  OS {
40      group oneOf {
41          Android,
42          Windows,
43          IOS
44      }
45  }
46
47  Applications {
```

```
48      group [2..3] {
49          AngryBirds,
50          Facebook,
51          Shazaam,
52          Skype
53      }
54  }
```

5 Traduction

Maintenant que l'on a expliqué les différents concepts, on peut transformer le langage *TVL* en *SMTLIBv2*. Pour ce faire, un exemple de chaque structure *TVL* est présenté ainsi que la traduction équivalente.

Remarques

Comme la version 2 de *SMT-LIB* est assez récente, les différents solveurs (cf. 2.3.1, p40) ne le supportent que partiellement. Il se peut donc que des erreurs apparaissent malgré l'emploi correct du langage.
Par exemple, la fonction *xor* sur les booléens ne fonctionne pas sur *z3* [28] ou *STP* [22] si l'on emploie la syntaxe avec plus de deux éléments. En revanche, cela fonctionne pour *and* et *or*, par exemple. Dans le listing 4, bien que les deux notations soient censées être équivalentes, la ligne 1 donne une erreur et la ligne 2 fonctionne.

Listing 4 – Erreur avec l'opérateur *xor*

```
1 (assert (=> OS (xor Android Windows IOS)))
2 (assert (=> OS (xor Android (xor Windows IOS))))
```

En ce qui concerne les différents exemples en *SMTLIBv2*, pour ne pas me répéter, j'ai omis la commande permettant de définir de la logique, la commande permettant de vérifier la satisfiabilité et la commande de terminaison. Pour pouvoir être accepté par le solveur et pour en recevoir une réponse, chaque bout de code *SMTLIBv2* doit être inclus en suivant la structure type présentée sur le listing 5.

Listing 5 – Structure type *SMT-LIBv2*

```
(set-logic QF_ABV)
... exemple ...
(check-sat)
(exit)
```

Suite à l'analyse, il apparaît qu'il y ait trois possibilités pour représenter des features :

1. Elles sont représentées par des booléens
2. Elles sont représentées par des *bitvectors* de taille 1
3. Elles sont représentées par des booléens et des *bitvectors* de taille 1

L'avantage de n'utiliser qu'un seul type de donnée est que cela facilite légèrement l'implémentation. En effet, il ne faut pas vérifier son type pour indiquer la syntaxe *SMT-LIBv2* correspondante à chaque appel d'une variable représentant une feature.

Dans l'exemple suivant, A est la feature parent et B est la sous-feature obligatoire. Chaque ligne correspond aux choix présentés ci-dessus. Par exemple, à la ligne 3, la feature A est représentée par un booléen et la feature B par un *bitvector* de taille 1.

```
1 (assert (=> A B))
2 (assert (=> (= A #b1) (= B #b1) ))
3 (assert (=> (= A #b1) B))
```

Dans notre traduction, certaines structures *TVL* n'ont pas été prises en compte. Il s'agit notamment du type d'attribut réel (*real*) qui est plus difficile à représenter vu l'utilisation de *bitvector* et du type d'attribut structuré (*struct*).

Conventions mathématiques

Pour les traductions suivantes, on emploie ces différentes notations lors des transformations génériques :

- La variable p représente le parent des sous-features ;
- L'ensemble $\mathbf{C}$ représente l'ensemble des sous-features de p ;
- L'ensemble $\mathbf{O}$ représente l'ensemble des sous-features optionnelles de p (donc $\mathbf{O} \in \mathbf{C}$).

5.1 allOf

Listing 6 – Traduction – **allOf** – *SMT-LIBv2*

```
1  root TabletPC {
2      group allOf {
3          Screen,
4          Processor,
5          Memory,
6          Camera,
7          OS,
8          opt Keyboard,
9          Applications
10     }
11 }
```

Transformation générique

$$\forall x \in \mathbf{C}: \quad \text{(declare-fun } x_i \text{ () Bool)}$$
$$\text{(assert(=> } x_i \ p))$$
$$\text{(assert (=> } p \text{ (and } x_1 \ \ldots \ x_n))),\ x \in \mathbf{C} \setminus \mathbf{O}$$

Tout d'abord, on déclare les différentes features en tant que booléen (*Bool*). Cela permet de les employer dans les différentes assertions. Pour cette traduction, je me base sur le tableau 1 représentant le mapping d'un *feature model* vers la logique des propositions [27, 10]. Il y apparaît que lorsqu'une feature est sélectionnée, son parent doit l'être également. Pour cette raison, *(assert(=> x_i p))* correspond à ces implications. Ensuite, on définit que leur conjonction (hors features optionnelles) doit toujours être vraie si la feature parent est sélectionnée.

Traduction

Listing 7 – Traduction – **allOf** – *SMT-LIBv2*

```
1  (declare-fun TabletPC () Bool)
2  (declare-fun Screen () Bool)
3  (declare-fun Processor () Bool)
4  (declare-fun Memory () Bool)
5  (declare-fun Camera () Bool)
6  (declare-fun OS () Bool)
7  (declare-fun Keyboard () Bool)
8  (declare-fun Applications () Bool)
9  (assert (=> TabletPC (and Screen Memory Camera OS
       Applications Processor)))
10 (assert (=> Screen TabletPC))
```

```
11 (assert (=> Memory TabletPC))
12 (assert (=> Camera TabletPC))
13 (assert (=> OS TabletPC))
14 (assert (=> Applications TabletPC))
15 (assert (=> Keyboard TabletPC))
16 (assert (=> Processor TabletPC))
```

5.2 oneOf

Listing 8 – Traduction – **oneOf** – *TVL*

```
1 OS {
2     group oneOf {
3         Android
4         Windows
5         IOS
6     }
7 }
```

Transformation générique

$$\forall x \in \mathbf{C} : \quad \text{(declare-fun } x_i \text{ () Bool)}$$
$$\text{(assert(=> } x_i \ p))$$
$$\text{(assert (=> } p \text{ (xor } x_1 \ \ldots \ x_n))), \ x \in \mathbf{C}$$

La traduction suit le même déroulement que celui présenté dans la section 5.1 (p67), si ce n'est que la conjonction est remplacée ici par un "ou exclusif" (*xor*) étant donné que l'on ne doit avoir qu'une seule sous-feature sélectionnée lorsque le parent est également sélectionné.

Traduction

Listing 9 – Traduction – **oneOf** – *SMT-LIBv2*

```
(declare-fun OS () Bool)
(declare-fun Android () Bool)
(declare-fun Windows () Bool)
(declare-fun IOS () Bool)
(assert (=> OS (xor Android Windows IOS)))
(assert (=> Android OS))
(assert (=> Windows OS))
(assert (=> IOS OS))
```

5.3 someOf

Listing 10 – Traduction – **someOf** – *TVL*

```
Extern {
    group someOf {
        SecureDigital
        MemoryStick
        MultiMediaCard
    }
}
```

Transformation générique

$$\begin{aligned} \forall x \in \mathbf{C} : \quad & (\text{declare-fun } x_i \text{ () Bool}) \\ & (\text{assert}(=> x_i\ p)) \\ (\text{assert } (=> p\ (\text{or } x_1\ \ldots\ x_n))),\ & x \in \mathbf{C} \end{aligned}$$

La traduction suit le même déroulement que celui présenté dans la section 5.1 (p67), si ce n'est que la conjonction est remplacée ici par un "ou" (*or*) étant donné que l'on ne doit avoir qu'au minimum une sous-feature sélectionnée lorsque le parent est également sélectionné.

Traduction

Listing 11 – Traduction – **someOf** – *SMT-LIBv2*

```
1 (declare-fun Extern () Bool)
2 (declare-fun SecureDigital () Bool)
3 (declare-fun MemoryStick () Bool)
4 (declare-fun CompactFlash () Bool)
5 (assert (=> Extern (or SecureDigital MemoryStick
      CompactFlash)))
6 (assert (=> SecureDigital Extern))
7 (assert (=> MemoryStick Extern))
8 (assert (=> CompactFlash Extern))
```

5.4 group [i..j]

Cette décomposition est plus complexe que les autres car il faut respecter les cardinalités i et j. Deux possibilités de traduction sont envisageables :

énumérer toutes les solutions possibles (1), calculer le nombre de features sélectionnées et voir si ce nombre est bien situé entre les bornes i et j (2).

Listing 12 – Traduction – **group** avec cardinalités – *TVL*

```
1 Applications {
2     group [2..3] {
3         AngryBirds
4         Facebook
5         Shazaam
6         Skype
7     }
8 }
```

5.4.1 Solution 1

Cette solution consiste à énumérer tous les ensembles possibles. Il s'agit donc d'une combinaison mathématique. Parmi un ensemble de n features, il faut créer tous les sous-ensembles contenant de i à j features. On doit également spécifier les sous-ensembles qui ne conviennent pas lors de la configuration du produit.

$$C_n^k = \frac{n!}{k!(n-k)!}$$

Dans notre cas, la borne inférieure (2) nous donne 6 possibilités qui sont présentées ci-dessous :

- AngryBirds, Facebook
- AngryBirds, Shazaam
- AngryBirds, Skype

- Facebook, Shazaam
- Facebook, Skype
- Shazaam, Skype

Le borne supérieure (3), quant à elle, nous donne 4 possibilités :

- AngryBirds, Facebook, Shazaam
- AngryBirds, Facebook, Skype
- AngryBirds, Shazaam, Skype
- Facebook, Shazaam, Skype

Transformation générique

Dans la formule suivante, $P_k(C)$ représente l'ensemble des k-combinaisons de $\mathbf{C}$. Pour rappel, une k-combinaison est une partie à k éléments de $\mathbf{C}$ [7]. L'ensemble des combinaisons possibles correspond donc à l'union des $P_k(\mathbf{C})$ où $i \leq k \leq j$, i représente la borne inférieure et j la borne supérieure. On a donc $\mathbf{E} = P_i(\mathbf{C}) \cup \ldots \cup P_j(\mathbf{C})$.

Dans la formule suivante, y correspond à l'ensemble des combinaisons possibles et est représenté sous la forme : $Z_1 \ldots Z_n$, où $\mathbf{Z} \in \mathbf{E}$ et chaque combinaison $\mathbf{Z}$ (formée de n éléments) est représentée de cette façon : (and a_1 ... a_n), $a \in \mathbf{Z}$.

$$\begin{aligned} \forall x \in \mathbf{C} : \quad & (\text{declare-fun } x_i \text{ () Bool}) \\ & (\text{assert}(=> x_i \ p)) \\ (\text{assert } (=> \ p \ (\text{or } y)) & \end{aligned}$$

Comme expliqué dans la traduction ci-dessous, il faudra également penser à citer les cas qui ne conviennent pas.

7. http ://fr.wikipedia.org/wiki/Combinaison_(mathématiques)

Traduction

Ci-dessous, la traduction du **group[2..3]**. La ligne *6* correspond à la borne inférieure *2* et la ligne *7* correspond à la borne supérieure *3*. La ligne *9*, quant à elle, correspond aux sous-ensembles qui ne peuvent pas se retrouver dans le produit. Dans ce cas, il s'agit seulement de la sélection de toutes les features de la relation (*AngryBirds*, *Facebook*, *Shazaam*, *Skype*).

Listing 13 – Traduction – 1 – **group** avec cardinalités – *SMT-LIBv2*

```
1  (declare-fun Applications () Bool)
2  (declare-fun AngryBirds () Bool)
3  (declare-fun Facebook () Bool)
4  (declare-fun Shazaam () Bool)
5  (declare-fun Skype () Bool)
6  (assert (=> AngryBirds Applications))
7  (assert (=> Facebook Applications))
8  (assert (=> Shazaam Applications))
9  (assert (=> Skype Applications))
10 (assert (=> Applications (or
11 (and Angrybirds Facebook) (and Angrybirds Shazaam
       ) (and Angrybirds Skype) (and Facebook Shazaam
       ) (and Facebook Skype) (and Shazaam Skype)
12 (and Angrybirds Facebook Shazaam) (and Angrybirds
        Facebook Skype) (and Angrybirds Shazaam Skype
       ) (and Facebook Shazaam Skype)
13 )))
14 (assert (not(and Angrybirds Facebook Shazaam
       Skype)))
```

5.4.2 Solution 2

Il faut donc, dans cette deuxième solution, comptabiliser le nombre total de features sélectionnées et vérifier que ce nombre respecte bien les contraintes i et j. Pour ce faire, on déclare les features en tant que *bitvectors* de taille 1. Un feature aura la valeur 1 (*#b1*) si celle-ci est sélectionnée, sinon 0 (*#b0*). Elles pourront être additionner dans le but de calculer de nombre de features sélectionnées.
Lors de l'implémentation, il faudra faire attention à vérifier le type de représentation des features car ils pourront être soit de type *bool*, soit de type *bitvector*.
On remarque également qu'avant de pouvoir additionner les différents *bitvectors*, on a besoin de les agrandir avec la commande *concat*.

```
(concat (_ BitVec i) (_ BitVec j))
```

Par exemple, (concat #b1 #b000) donne #b1000.

Dans notre le cas, comme le nombre d'enfants s'élève à 4, il faut 3 bits pour le représenter (*bitvector* non-signé). On ajoute donc 2 bits à chacune des features avant de les additionner.

Transformation générique

$$\begin{aligned} \forall x \in \mathbf{C}: \quad & \text{(declare-fun } x_i \text{ () (\_ BitVec 1)} \\ & \text{(assert(=> (= } x_i \text{ \#b1) } p\text{))} \\ \text{(assert (let (sum y) z)} & \end{aligned}$$

où y représente la sommes des sous-features et z représente les contraintes.

On peut représenter y par :
(bvadd (concat (_bv0 n) x_1) (bvadd (concat (_bv0 n) x_2) ...)), où $x \in \mathbf{C}$ et $n = |C| - 1$.

Traduction

Listing 14 – Traduction – 2 – **group** avec cardinalités – *SMT-LIBv2*

```
(declare-fun Applications () Bool)
(declare-fun AngryBirds () (_ BitVec 1))
(declare-fun Facebook () (_ BitVec 1))
(declare-fun Shazaam () (_ BitVec 1))
(declare-fun Skype () (_ BitVec 1))
(assert (=> (= AngryBirds #b1) Applications))
(assert (=> (= Facebook #b1) Applications))
(assert (=> (= Shazaam #b1) Applications))
(assert (=> (= Skype #b1) Applications))
(assert (let (sum (bvadd (concat (_ bv0 2)
    AngryBirds) (bvadd (concat (_ bv0 2) Facebook)
     (bvadd (concat (_ bv0 2) Shazaam) (concat (_
    bv0 2) Skype))))) (=> Applications (and (bvuge
     sum (_ bv2 3)) (bvule sum (_ bv3 3)))) ))
```

La commande **let** permet de définir une ou plusieurs variables locales. Dans cet exemple, *sum* représente le nombre de features sélectionnées. Pour rappel, une feature sélectionnée vaut *#b1*.

```
(let ((x1 t1 ) ...(xn tn )) t)
```

5.5 Contraintes classiques

Dans le tableau 3, on présente la traduction des contraintes classiques (*requires* et *excludes*) en *TVL* vers *SMT-LIBv2*. Ici, les F représentent des features.

Description	**Syntaxe *TVL***	**Syntaxe *SMT-LIBv2***
Requires	$F1$ requires F_2	$(=> F_1\ F_2)$
Excludes	$F1$ requires F_2	(not (and $F_1\ F_2$))

TABLE 3 – Mapping des contraintes classiques en *TVL* vers *SMT-LIBv2*

En se basant sur le tableau 3, on peut appliquer les différentes règles sur l'exemple illustré dans le listing 15. Sa traduction sera présentée sur le listing 16. Il ne faut évidemment pas oublier de déclarer les différentes variables (dans ce cas, MemoryStick – Ligne 1 et CompactFlash – Ligne 2) pour pouvoir vérifier les différentes assertions.

Listing 15 – Traduction – Contrainte *cross-tree* classique – *TVL*

```
1 MemoryStick excludes CompactFlash;
```

Listing 16 – Traduction – Contrainte *cross-tree* classique – *SMT-LIBv2*

```
1 (declare-fun MemoryStick () Bool)
2 (declare-fun CompactFlash () Bool)
3 (assert (not (and MemoryStick CompactFlash)))
```

5.6 Logique des propositions

Comme nous l'avons vu précédemment (2.1), les contraintes sur un *feature model* peuvent être définies en logique des propositions. Sur le tableau 4 et 5, on va définir la correspondance entre la déclaration de contraintes booléennes en *TVL* et son équivalence en *SMT-LIBv2*.

Dans notre tableau, chaque E représente une expression. Effectivement, toute expression peut en contenir une autre.
Regardons l'exemple en *TVL* illustré sur le listing 17. La contrainte est composée d'une expression globale d'implication ($->$). Cette expression est alors divisée en sous-expresssions : celle de gauche représente une feature et celle de droite une aggrégation "et" (*and*), et ainsi de suite.

Description	**Symb.**	**Syntaxe *TVL***	**Syntaxe *SMT-LIBv2***
Conjonction	$\wedge$	E_1 && E_2	(and E_1 E_2)
Disjonction	$\vee$	$E_1 \| \| E_2$	(or E_1 E_2)
Egalité	$=$	$E_1 = E_2$	($=$ E_1 E_2)
Implication	$\rightarrow$	$E_1 -> E_2$	($=>$ E_1 E_2)
Equivalence	$\leftrightarrow$	$E_1 <-> E_2$	(and ($=>$ E_1 E_2) ($=>$ E_2 E_1))
Négation	$\neg$	! E	(not E)

TABLE 4 – Mapping des expressions booléennes en *TVL* vers *SMT-LIBv2*

En suivant ces différentes règles de transformation, on peut désormais traduire le code *TVL* (Listing 17) vers le code *SMT-LIBv2* (Listing 18).

Listing 17 – Traduction – Exemple de contraintes – *TVL*

```
1 IOS -> and(Front,Back)
```

Description	**Syntaxe *TVL***	**Syntaxe *SMT-LIBv2***
Conjonction	and(E_1, E_2, ..., E_n)	(and E_1 E_2 ... E_n)
Disjonction	or(E_1, E_2, ..., E_n)	(or E_1 E_2 ... E_n)
Ou exclusif	xor(E_1, E_2, ..., E_n)	(xor E_1 E_2 ... E_n)

TABLE 5 – Mapping des aggrégations booléennes en *TVL* vers *SMT-LIBv2*

Listing 18 – Traduction – Exemple de contraintes – *SMT-LIBv2*

```
1 (assert (=> High (and Manual Electric)))
```

5.7 Arithmétique

5.7.1 Opérateurs de base

Dans cette section, on présente la correspondance entre les opérateurs classiques tels que l'addition et la division en *TVL* vers leur équivalent en *SMT-LIBv2*. Etant donné que l'on travaille sur des entiers, la division sera de type entière. Par exemple, 8/7 vaut 1.

Description	**Symbole *TVL***	**Syntaxe *TVL***	**Syntaxe *SMT-LIBv2***
Addition	+	$E_1 + E_n$	(bvadd E_1 E_2)
Soustraction	−	$E_1 - E_2$	(bvurem E_1 E_2)
Multiplication	∗	$E_1 * E_2$	(bvmul E_1 E_2)
Division	/	E_1/E_2	(bvsdiv E_1 E_2)

TABLE 6 – Mapping des comparaisons en *TVL* vers *SMT-LIBv2*

5.7.2 Comparaison

Dans ce point, on crée une correspondance entre les opérateurs de comparaison en *TVL* et leur traduction en *SMT-LIBv2*. Les expressions E correspondent à des expressions entières (*int*) en *TVL*. Cela peut donc être une valeur numérique, une addition de valeurs numériques, ... Rappelons que l'on a choisi de représenter les entiers en vecteurs de bits. Pour cette raison, on emploie les différentes fonctions présentes dans la logique *QF_BV* (cf. 3.2.5, p54).

Description	**Syntaxe *TVL***	**Syntaxe *SMT-LIBv2***
$<$	$E_1 < E_n)$	(bvslt E_1 E_2)
$\leq$	$E_1 <= E_2$	(bvsle E_1 E_2)
$>$	$E_1 > E_2$	(bvsgt E_1 E_2)
$\geq$	$E_1 >= E_2$	(bvsge E_1 E_2)
$=$	$E_1 == E_2$	$(= E_1 E_2)$

TABLE 7 – Mapping des comparaisons en *TVL* vers *SMT-LIBv2*

5.8 Attributs

Avant d'employer un attribut, il doit être défini. Voici les différentes syntaxes en fonction du type d'attribut. Les entiers sont représentés comme des *bitvectors*, les booléens comme des *bool*.

```
(declare-fun int () (_ BitVec 32))
(declare-fun bool () Bool)
```

5.8.1 Attribut entier avec restriction

Comme on peut le constater dans l'exemple (Listing 19), un écran a un attribut **size** compris entre 6 et 10.

Listing 19 – Traduction – Attribut entier avec restriction – *TVL*

```
1 int size in [6..10];
```

Traduction

Lorsque que l'on a une restriction de domaine sur un entier, on emploie les prédicats binaires de comparaison (voir tableau 2 [5]).
Ici (Listing 20), la ligne 1 déclare la variable **size** de l'écran et la ligne 2 y ajoute les contraintes. En effet, **bvuge** va impliquer que **size** soit plus grand ou égal à 6, et **bvule** va impliquer que **size** soit plus petit ou égal à 10. Notons que l'on a employé les opérateurs de comparaison pour les entiers non-signés . Il faudrait par conséquent les changer si au moins une des bornes est négative.

Listing 20 – Traduction – Attribut entier avec restriction – *SMT-LIBv2*

```
1 (declare-fun size () (_ BitVec 32))
2 (assert (and (bvuge size (_ bv6 32)) (bvule size
    (_ bv10 32))))
```

5.8.2 Enumération

Listing 21 – Traduction – Enumération – *TVL*

```
1 enum resolution in {360p, 480p, 720p, 1080p};
```

Traduction

En ce qui concerne, l'énumération (*enum*), on a repris la solution proposée dans l'implémentation de base de *TVL*[8]. La traduction consiste à transformer les différents éléments de l'énumération en booléens et à faire en sorte que, parmi eux, un seul peut être sélectionné. Pour cela, on emploie le "ou exclusif".

Listing 22 – Traduction – Enumération – *SMT-LIBv2*

```
1 (declare-fun resolution_360p () Bool)
2 (declare-fun resolution_480p () Bool)
3 (declare-fun resolution_720p () Bool)
4 (declare-fun resolution_1080p () Bool)
5 (assert (xor resolution_360p resolution_480p
      resolution_720p resolution_1080p))
```

8. http ://www.info.fundp.ac.be/∼acs/tvl/

6 Implémentation

Pour l'implémentation, je suis parti d'un outil disponible sur le site web de *TVL*[9]. Cet outil implémenté en *JAVA* dispose de deux composants. Le composant syntaxique est un parseur implémenté par le générateur de parseur *CUP*[10]. Il permet de vérifier les types, les références ainsi que l'utilisation correcte de la syntaxe. Le composant sémantique implémente, quant à lui, la sémantique de *TVL* et permet de transformer un *feature model TVL* en une formule CNF qui sera vérifiée par la suite en utilisant *Sat4J*[11] [13].

Etant donné que le parseur et la représentation du *feature model* étaient pertinents, cette partie a été gardée. Par conséquent, j'ai étendu la partie sémantique afin de génerer un fichier *SMT-LIBv2*.

Voici les différentes étapes nécessaires au passage d'un *feature model* représenté en *TVL* vers sa traduction en *SMT-LIBv2*. Premièrement, on donne un fichier *TVL* en entrée au parseur. Celui-ci vérifie que la syntaxe est correcte comme expliqué précédemment. Grâce au parseur, les différents éléments représentant le modèle sont traduits sous forme d'objets qui permettent d'effectuer la transformation vers le langage *SMT-LIBv2*. Cette transformation suit les différentes règles de la section 5. Une fois la transformation finie, on a un fichier .smt2 contenant une représentation du *feature model* dans le langage *SMT-LIBv2*. Par la suite, ce fichier sera donné au solveur *SMT* de notre choix qui vérifiera la satisfiabilité du modèle. Ce flux de processus est illustré sur la figure 12.

9. http ://www.info.fundp.ac.be/∼acs/tvl
10. http ://www2.cs.tum.edu/projects/cup
11. http ://www.sat4j.org/

Durant notre étude, pour vérifier le langage au format *SMT-LIBv2*, on a principalement utilisé *z3* [28] et *STP* [22].

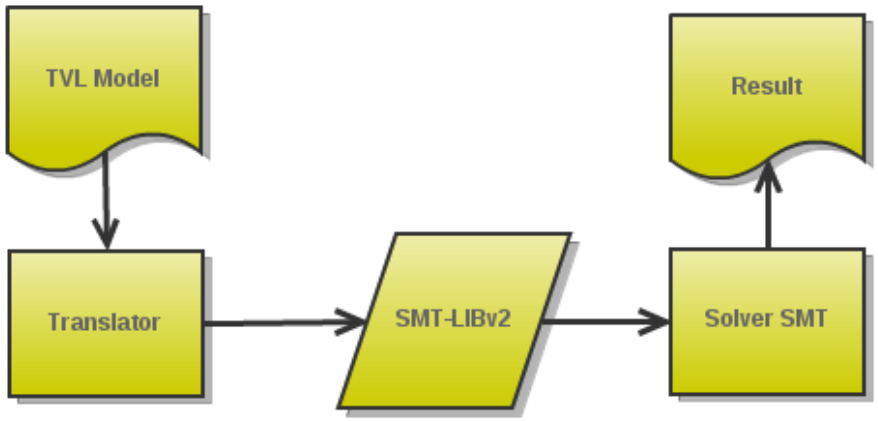


FIGURE 12 – Flux des processus

7 Conclusion

En consultant la littérature, j'ai remarqué que les notations pour les *feature models* étaient riches et variées ; parmi elles, j'ai choisi *TVL* pour son expressivité ainsi que sa capacité à représenter plus facilement des *feature models* assez complexes.

Ensuite, j'ai traité du domaine de la satifiabilité. J'ai comparé succinctement les différents solveurs ainsi que leur fonctionnement. Puis, il était question des problèmes *SMT* et l'on remarque que les recherches effectuées sur la traduction de *feature models* vers ce langage étaient peu nombreuses. Par conséquent, l'intérêt du document et de la traduction s'en sont vu renforcés.

Avec tous les éléments en main, j'ai proposé une traduction. Premièrement, différents travaux déjà réalisés sur la question de la satisfiabilité du *feature model* m'ont permis de baliser le travail et m'ont aidé à trouver des solutions aux divers problèmes rencontrés. Citons par exemple, le mapping d'un *feature model* vers la logique des propositions utilisé dans divers travaux.
Un des avantages de l'utilisation des solveurs *SMT* est qu'ils emploient des théories spécifiques pour résoudre des problèmes où les variables ne sont plus forcément binaires. Grâce à cela, on peut, par exemple, tenir compte du type entier dans cette traduction.

Durant la traduction, j'ai essayé de rester le plus générique possible. Ceci permettra, à celui qui le désire, de disposer de règles de traduction, tout en lui laissant la liberté des choix de l'implémentation.

Pour finir, bien que le nombre de tests fût limité, cette approche, qui consiste en l'utilisation d'un langage *SMT* pour effectuer des analyses automatiques de *feature models*, s'est avérée prometteuse. Elle semble être une alternative viable aux approches utilisant un solveur SAT, *BDD* ou *CSP*.

7.1 Travaux futurs

Dans ce qui suit, on détaille quelques travaux futurs qui pourraient compléter cette étude.

- Etant donné que la traduction de *TVL* vers *SMT-LIBv2* n'est pas complète, il serait intéressant de traduire les structures manquantes. On peut citer, par exemple, le type d'attribut réel (*real*) ou le type d'attribut structuré (*struct*).
- Etant donné le nombre restreint de tests, on pourrait imaginer appliquer notre traduction sur un ensemble plus large de *feature models* en *TVL*. Ceci permettrait de vérifier le bon fonctionnement de l'implémentation développée et ses performances.
- Il serait également judicieux d'effectuer différents *benchmarks*. En effet, on pourrait comparer, par exemple, les résultats obtenus avec l'implémentation de référence (fournie sur le site web de *TVL* [13]) avec la traduction en *SMT-LIBv2* présentée dans ce document. Il faudrait, pour ce faire, veiller à employer des mécanismes implémentés par les deux solutions.
- Il serait peut-être également intéressant de nous baser sur d'autres théories *SMT* lors de la traduction : par exemple, la théorie des entiers et réels (*int* et *real*).

Références

[1] Henrik Reif Andersen. An introduction to binary decision diagrams. Lecture notes for Efficient Algorithms and Programs, 1999.

[2] Michal Antkiewicz. Concept Modeling Using Clafer, 2010. Draft version 5.

[3] Kacper Bak, Krzysztof Czarnecki, and Andrzej Wasowski. Feature and meta-models in clafer : Mixed, specialized, and coupled. In 3rd International Conference on Software Language Engineering, nov 2010.

[4] Clark Barrett, Aaron Stump, and Cesare Tinelli. The smt-lib standard version 2.0. 2010.

[5] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). WebSite, 2010. `www.SMT-LIB.org`.

[6] Don Batory. Feature models, grammars, and propositional formulas. In 9th International Software Product Lines Conference (SPLC 2005), volume 3714, pages 7–20. Springer, 2005.

[7] Don Batory, David Benavides, and Antonio Ruiz-Cortes. Automated analysis of feature models : challenges ahead. Commun. ACM, 49 :45–47, December 2006.

[8] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. LNCS, Advanced Information Systems Engineering : 17th International Conference, CAiSE 2005, 3520 :491–503, 2005.

[9] D. Benavides, A. Ruiz-Cortés, P. Trinidad, and S. Segura. A survey on the automated analyses of feture models. In Jornadas de Ingeniería del Software y Bases de Datos (JISBD), 2006.

[10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later : a literature review. Information Systems, 35(6), 2010.

[11] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. Handbook of Satisfiability : Volume 185 Frontiers in Artificial Intelligence and Applications. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.

[12] A. Classen, Q. Boucher, P. Faber, and P. Heymans. Syntax and semantics of tvl, a text-based feature modelling language. Technical report, FUNDP - MoVeS, jan 2010.

[13] Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling : Syntax and semantics of TVL. Science of Computer Programming, Special Issue on Software Evolution, Adaptability and Variability, 76(12) :1130–1143, nov 2011.

[14] David R. Cok. The smt-libv2 language and tools : A tutorial. 2011.

[15] Krzysztof Czarnecki and Ulrich Eisenecker. Generative Programming : Methods, Tools, and Applications. Addison–Wesley, may 2000.

[16] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration using feature models. In Software Product Lines : Third International Conference (SPLC 2004), pages 266–283. Springer-Verlag, sep 2004.

[17] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. Software Process : Improvement and Practice, 10, jan 2005.

[18] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration through specialization and multi-level configuration of

feature models. In Software Process : Improvement and Practice, pages 143–169, apr 2005.

[19] Krzysztof Czarnecki and Chang Hwan Peter Kim. Cardinality-based feature modeling and constraints : a progress report. In International Workshop on Software Factories at OOPSLA'05, San Diego, California, USA, 2005. ACM, ACM.

[20] Mike Dean and Guus Schreiber. Owl web ontology language reference. WebSite, feb 2004. `http://www.w3.org/TR/2004/REC-owl-ref-20040210/`.

[21] Vijay Ganesh. Stp constraint solver. WebSite. `http://sites.google.com/site/stpfastprover/STP-Fast-Prover`.

[22] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. CAV 2007 (LNCS 4590), pages 524–536, 2007.

[23] Martin L. Griss, John Favaro, and Case Methodologist. Integrating feature modeling with the rseb. In In Proceedings of the Fifth International Conference on Software Reuse, pages 76–85, 1998.

[24] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Software Engineering Institute, Carnegie Mellon Universitys, nov 1990.

[25] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Gerard Jounghyun Kim, and Euiseob Shin. Form : A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering, 5 :143–168, 1998.

[26] Joseph Kiniry. Reasoning about feature models in higher-order logic. In Proceedings of the 11th International Software Product Line Conference, SPLC '07. IEEE Computer Society, 2007.

[27] Marcilio Mendonca, Andrzej Wasowski, and Krzysztof Czarnecki. Sat-based analysis of feature models is easy. In 13th International Software Product Lines Conference (SPLC 2009), pages 231–240, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[28] Microsoft. Solveur z3. WebSite, 2011. `http://research.microsoft.com/en-us/um/redmond/projects/z3/index.html`.

[29] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff : engineering an efficient sat solver. In Proceedings of the 38th annual Design Automation Conference, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[30] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories : From an abstract davis–putnam–logemann–loveland procedure to dpll(t). Journal of the ACM, 53(6) :937–977, 2006.

[31] Silvio Ranise, Cesare Tinelli, and Clark Barrett. Logic qf_bv, 2011. `http://goedel.cs.uiowa.edu/smtlib/logics/QF_BV.smt2`.

[32] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow. Extending feature diagrams with UML multiplicities, pages 1–7. Citeseer, 2002.

[33] Jose Evelio Martinez Saiz. Feature models visualization based on ontology framework. Master's thesis, Vrije Universiteit Brussel, 2009.

[34] Pierre-Yves Schobbens, Patrick Heymans, Jean christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. Computer Networks and Isdn Systems, 51 :456–479, 2007.

[35] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature diagrams : A survey and a formal semantics.

Requirements Engineering, IEEE International Conference on, 0 :139–148, 2006.

[36] Hai Wang, Yuan Fang Li, Jing Sun, Hongyu Zhang, and Jeff Pan. A semantic web approach to feature modeling and verification. Workshop on Semantic Web Enabled Software Engineering (SWESE'05), nov 2005.

[37] Wikipedia. Binary decision diagram. WebSite, 2011. `http://en.wikipedia.org/wiki/Binary_decision_diagram`.

[38] Wikipedia. Boolean satisfiability problem. WebSite, 2011. `http://en.wikipedia.org/wiki/Propositional_satisfiability`.

[39] Wikipedia. Constraint satisfaction problem. WebSite, 2011. `http://en.wikipedia.org/wiki/Constraint_satisfaction_problem`.

[40] Wikipedia. Dpll algorithm. WebSite, 2011. `http://en.wikipedia.org/wiki/DPLL_algorithm`.

[41] Wikipedia. Feature model. WebSite, 2011. `http://en.wikipedia.org/wiki/Feature_model`.

[42] Wikipedia. Inference engine. WebSite, 2011. `http://en.wikipedia.org/wiki/Inference_engine`.

[43] Wikipedia. Np-complete. WebSite, 2011. `http://en.wikipedia.org/wiki/Np-complete`.

[44] Wikipedia. Satisfiability modulo theories. WebSite, 2011. `http://en.wikipedia.org/wiki/Satisfiability_Modulo_Theories`.